Artificial Software Diversification for WebAssembly

JAVIER CABRERA-ARTEAGA

Doctoral Thesis
Supervised by
Benoit Baudry and Martin Monperrus
Stockholm, Sweden, 2023

KTH Royal Institute of Technology
School of Electrical Engineering and Computer Science
Division of Software and Computer Systems
SE-10044 Stockholm
Sweden

TRITA-EECS-AVL-2020:4
ISBN 100-

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av Teknologie doktorexamen i elektroteknik i .

Tryck: Universitetsservice US AB

Abstract

[1]

Keywords: Lorem, Ipsum, Dolor, Sit, Amet

Sammanfattning

[1]

LIST OF PAPERS

1. ***WebAssembly Diversification for Malware Evasion***
   Javier Cabrera-Arteaga,Tim Toady, Martin Monperrus, Benoit Baudry
   Computers & Security, Volume 131, 2023
   https://www.sciencedirect.com/science/article/pii/S0167404823002067

2. ***Wasm-mutate: Fast and Effective Binary Diversification for WebAssembly***
   Javier Cabrera-Arteaga, Nicholas Fitzgerald, Martin Monperrus, Benoit Baudry

3. ***Multi-Variant Execution at the Edge***
   Javier Cabrera-Arteaga,Pierre Laperdrix, Martin Monperrus, Benoit Baudry
   Conference on Computer and Communications Security (CCS 2022), Moving Target Defense (MTD)
   https://dl.acm.org/doi/abs/10.1145/3560828.3564007

4. ***CROW: Code Diversification for WebAssembly***
   Javier Cabrera-Arteaga, Orestis Floros, Oscar Vera-Pérez, Benoit Baudry, Martin Monperrus
   Network and Distributed System Security Symposium (NDSS 2021), MADWeb
   https://doi.org/10.14722/madweb.2021.23004

5. ***Superoptimization of WebAssembly Bytecode***
   Javier Cabrera-Arteaga,Shrinish Donde, Jian Gu, Orestis Floros, Lucas Satabin, Benoit Baudry, Martin Monperrus
   Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming (Programming 2021), MoreVMs
   https://doi.org/10.1145/3397537.3397567

6. ***Scalable Comparison of JavaScript V8 Bytecode Traces***
   Javier Cabrera-Arteaga,Martin Monperrus, Benoit Baudry
   11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (SPLASH 2019)
   https://doi.org/10.1145/3358504.3361228

ACKNOWLEDGEMENT

ACRONYMS

List of commonly used acronyms:

Wasm WebAssembly

Contents

Part I

Thesis

01 INTRODUCTION

TODO Recent papers first. Mention Workshops instead in conference. "Procedings of XXXX". Add the pages in the papers list.

1.1 Background

TODO Motivate with the open challenges.

1.2 Problem statement

TODO Problem statement **TODO** Set the requirements as R1, R2, then map each contribution to them.

1.3 Automatic Software diversification requirements

1. **1:** **TODO** Requirement 1

1.4 List of contributions

C1: Methodology contribution: We propose a methodology for generating software diversification for WebAssembly and the assessment of the generated diversity.

C2: Theoretical contribution: We propose theoretical foundation in order to improve Software Diversification for WebAssembly.

C3: Automatic diversity generation for WebAssembly: We generate WebAssembly program variants.

C4: Software Diversity for Defensive Purposes: We assess how generated WebAssembly program variants could be used for defensive purposes.

C5: Software Diversity for Offensives Purposes: We assess how generated WebAssembly program variants could be used for offensive purposes, yet improving security systems.

Contribution	Resarch papers				
	P1	P2	P3	P4	P5
C1	x	x		x	x
C2	x	x			
C3	x	x	x		
C4	x	x	x		
C5			x		
C6	x	x	x	x	x

Table 1.1: Mapping of the contributions to the research papers appended to this thesis.

C6: Software Artifacts: We provide software artifacts for the research community to reproduce our results.

TODO Make multi column table

1.5 Summary of research papers

P1: Superoptimization of WebAssembly Bytecode.

P2: CROW: Code randomization for WebAssembly bytecode.

P3: Multivariant execution at the Edge.

P4: Wasm-mutate: Fast and efficient software diversification for WebAssembly.

P5: WebAssembly Diversification for Malware evasion.

1.6 Thesis outline

02 BACKGROUND AND STATE OF THE ART

2.1 WebAssembly

The W3C publicly announced the WebAssembly language in 2017 as the four scripting language supported in all major web browser vendors. Wasm is a binary instruction format for a stack-based virtual machine and was officially consolidated by the work of Haas et al. [1] in 2017. Wasm is designed to be fast, portable, self-contained and secure, and it promises to outperform JavaScript execution [1].

Since 2017, the adoption of Wasm keeps growing. For example; Adobe, announced a full online version of Photoshop[1] written in WebAssembly; game companies moved their development from JavaScript to Wasm like is the case of a full Minecraft version[2]. Moreover, WebAssembly has been evolving outside web browsers since its first announcement. Some works demonstrated that using WebAssembly as an intermediate layer is better in terms of startup and memory usage than containerization and virtualization [2, 3]. Consequently, in 2019, the Bytecode Alliance [4] proposed WebAssembly System Interface (WASI) [5]. WASI pioneered the execution of Wasm with a POSIX system interface protocol, making possible to execute Wasm directly in the operating system. Therefore, it standardizes the adoption of Wasm in heterogeneous platforms [6], making it suitable standalone and backend execution scenarios [7, 8].

2.1.2 WebAssembly's generation

WebAssembly programs are pre-compiled from source languages like C/C++, Rust, or Go, which means that it can benefit from the optimizations of the source language compiler. The resulting Wasm program is like a traditional shared library, containing instruction codes, symbols, and exported functions. A host environment is in charge of complementing the Wasm program, such as providing external functions required for execution within the host engine. For instance, functions for interacting with an HTML page's DOM are imported into the Wasm binary when invoked from JavaScript code.

[1] `https://twitter.com/Adobe/status/1453034805004685313?s=20&t=Zf1N7-WmzecA0K4V8R69lw`

[2] `https://satoshinm.github.io/NetCraft/`

TODO Expand on how we generate it

In Listing 2.1 and Listing 2.2, we illustrate a C program and its corresponding Wasm binary. The C function includes heap allocation, external function usage, and a function definition featuring a loop, conditional branching, function calls, and memory accesses. The Wasm code in Listing 2.2 displays the textual format of the generated Wasm (Wat).

```c
// Some raw data
const int A[250];

// Imported function
int ftoi(float a);

int main() {
    for(int i = 0; i < 250; i++) {
        if (A[i] > 100)
            return A[i] + ftoi(12.54);
    }

    return A[0];
}
```

Listing 2.1: Example C program which includes heap allocation, external function usage, and a function definition featuring a loop, conditional branching, function calls, and memory accesses.

```
; WebAssembly magic bytes(\0asm) and version (1.0) ;
(module
; Type section: 01 ... ;
  (type (;0;) (func (param f32) (result i32)))
  (type (;1;) (func))
  (type (;2;) (func (result i32)))
; Import section: 02 ... ;
  (import "env" "ftoi" (func $ftoi (type 0)))
; Code section: 03 ... ;
  (func $main (type 2) (result i32)
    (local i32 i32)
    i32.const -1000
    local.set 0
    block  ;label = @1;
      loop  ;label = @2;
        i32.const 0
        local.get 0
        i32.add
        i32.load
        local.tee 1
        i32.const 101
        i32.ge_s
        br_if 1 ;@1;
        local.get 0
        i32.const 4
        i32.add
        local.tee 0
        br_if 0 ;@2;
      end
      i32.const 0
      return
    end
    f32.const 0x1.9147aep+3
    call $ftoi
    local.get 1
    i32.add)
; Memory section: 05 ... ;
  (memory (;0;) 1)
; Global section: 06 ... ;
  (global (;4;) i32 (i32.const 1000))
; Export section: 07 ... ;
  (export "memory" (memory 0))
  (export "A" (global 2))
; Data section: 0d ... ;
  (data $data (0) "\00\00\00\00...")
)
```

Listing 2.2: Wasm code for Listing 2.1. The example Wasm code illustrates the translation from C to Wasm in which several high-level language features are translated into multiple Wasm instructions.

2.1.3 WebAssembly's binary format

The Wasm binary format is close to machine code and already optimized. Thus, its consuming process typically involves a straightforward one-to-one mapping. For example, a compiler might accelerate the compilation process by parallelizing the parsing process. The main reason behind this claim is that a Wasm binary is organized into a contiguous collection of sections. In Figure 2.1 we show the binary format of a Wasm section. A Wasm section starts with a 1-byte section ID, followed by an 8-byte section size, and concludes with the section content, which precisely matches the size indicated earlier.

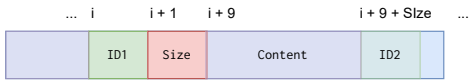


Figure 2.1: Memory byte representation of a WebAssembly binary section, starting with a 1-byte section ID, followed by an 8-byte section size, and finally the section content.

A Wasm binary contains sections of 12 types, each with a specific semantic role and placement within the module. Each section is optional, where an omitted section is considered empty. In the following text, we summarize each one of the 12 types of Wasm sections, providing their name, ID, and purpose. In addition, some sections are annotated as comments in the Wasm code in Listing 2.2.

TODO Check if better with examples in each section

Custom Section (`00`) : Comprises two parts: the section name and arbitrary content. Primarily used for storing metadata, such as the compiler used to generate the binary. This type of section has no order constraints with other sections and is optional. Compilers usually skip this section when consuming a WebAssembly binary.

Type Section (`01`) : Contains the function signatures for functions declared or defined within the binary. It must occur only once in a binary. It can be empty.

Import Section (`02`) : Lists elements imported from the host, including functions, memories, globals, and tables. It must occur only once in a binary. It can be empty.

Function Section (`03`) : Details functions defined within the binary. It essentially maps Type section entries to Code section entries. It must occur only once in a binary. It can be empty.

Table Section (`04`) : Groups functions with identical signatures to control indirect calls. It must occur only once in a binary. It can be empty.

Memory Section (05) : Specifies the number and initial size of unmanaged linear memories. It must occur only once in a binary. It can be empty.

Global Section (06) : Defines global variables as managed memory for use and sharing between functions in the WebAssembly binary. It must occur only once in a binary. It can be empty.

Export Section (07) : Declares elements like functions, globals, memories, and tables for host engine access. The entry point of the WebAssembly binary is typically declared here. It must occur only once in a binary. It can be empty.

Start Section (08) : Designates a function to be called upon binary readiness, initializing the WebAssembly program state before executing any exported functions. It must occur only once in a binary. It can be empty.

Element Section (09) : Contains elements to initialize the binary tables. It must occur only once in a binary. It can be empty.

Code Section (10) : Contains the body of functions defined in the Function section. Each entry consists of local variables used and a list of instructions. It must occur only once in a binary. It can be empty.

Data Section (11) : Holds data for initializing unmanaged linear memory. Each entry specifies the offset and data to be placed in memoryIt must occur only once in a binary. It can be empty.

Data Count Section (12) : Primarily used for validating the Data Section. If the segment count in the Data Section mismatches the Data Count, the binary is considered malformed. It must occur only once in a binary. It can be empty.

2.1.4 WebAssembly's Runtime structure

TODO Reorder this text according with the wrules

The WebAssembly runtime structure is described in the WebAssembly specification by enunciating 10 key components: the Store, Stack, Locals, Module Instances, Function Instances, Table Instances, Memory Instances, Global Instances, Export Instances, and Import Instances. These components are particularly significant in maintaining the state of a WebAssembly program during its execution. In the following text, we provide a brief description of each runtime component. Notice that, the runtime structure is an abstraction that serves to validate Wasm consumers.

Store : The WebAssembly store represents the global state and is a collection of instances of functions, tables, memories, and globals. Each of these instances is uniquely identified by an address, which is usually represented as an i32 integer.

Module Instances : A module instance is a runtime representation of a loaded and initialized WebAssembly module. It contains the runtime representation of all the definitions within a module, including functions, tables, memories, and globals, as well as the module's exports and imports.

Table instances : A table instance is a vector of function elements. WebAssembly tables are used to support indirect function calls. For example, it allows modeling dynamic calls of functions (through pointers) from languages such as C/C++, for which the Wasm's compiler is in charge of populating the static table of functions.

Export Instances : Export instances represent the functions, tables, elements, globals or memories that are exported by a module.

Import Instances : Import instances represent the functions, tables, elements, globals or memories that are imported into a module from the host.

The Stack holds typed values and control frames, with control frames handling block instructions, loops, and function calls. Values inside the stack can be of the only static types allowed in Wasm 1.0, `i32` for 32 bits signed integer, `i64` for 64 bits signed integer, `f32` for 32 bits float and `f64` for 64 bits float. Therefore, abstract types, such as classes, objects, and arrays, are not natively supported. Instead, during compilation, such types are transformed into primitive types and stored in the linear memory.

Memory Instances represent the unmanaged linear memory of a WebAssembly program, consisting of a contiguous array of bytes. Memory instances are accessed with `i32` pointers (integer of 32 bits). Memory instances are usually bound in browser engines to 4Gb of size, and it is only shareable between the process that instantiates the WebAssembly module and the binary itself.

Global Instances : A global instance is a global variable with a value and a mutability flag, indicating whether the global can be modified or is immutable. Global variables are part of the managed data, i.e., their allocation and memory placement are managed by the host engine. Global variables are only accessible by their declaration index, and it is not possible to dynamically address them.

Locals : Locals are mutable variables that are local to a specific function invocation. As globals, locals are part of the managed data.

Note 1. *Along with this dissertation, as the work of Lehmann et al. [9], we refer to managed and unmanaged data to differentiate between the data that is managed by the host engine and the data that is managed by the WebAssembly program respectively.*

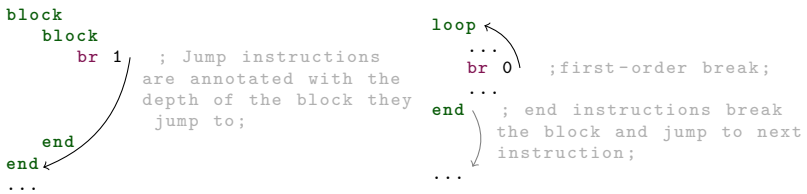
Function Instances : A function instance groups locals and a function body.

Locals are typed variables that are local to a specific function invocation. The function body is a sequence of instructions that are executed when the function is called. Each instruction either reads from the stack, writes to the stack, or modifies the control flow of the function. Recalling the example Wasm binary previously showed, the local variable declarations and typed instructions that are evaluated using the stack can be appreciated between Line 7 and Line 32 in Listing 2.2. Each instruction reads its operands from the stack and pushes back the result. In the case of Listing 2.2, the result value of the main function is the calculation of the last instruction, `i32.add` at result. As the listing shows, instructions are annotated with a numeric type.

2.1.5 WebAssembly's control flow

In WebAssembly, a defined function instructions are organized into blocks, with the function's starting point serving as the root block. Unlike traditional assembly code, control flow structures in Wasm jump between block boundaries rather than arbitrary positions within the code. Each block might specify the required stack state before execution and the resulting stack state after its instructions have run. This stack state is used to validate the binary during compilation and to ensure that the stack is in a valid state before executing the block's instructions. Blocks in Wasm are explicit, indicating, where they start and end. By design, each block cannot reference or execute code from outer blocks.

Control flow within a function is managed through three types of break instructions: unconditional break, conditional break, and table break. Importantly, each break instruction is limited to jumping to one of its enclosing blocks. Unlike standard blocks, where breaks jump to the end of the block, breaks within a loop block jump to the block's beginning, effectively restarting the loop. To illustrate this, Listing 2.3 provides an example comparing a standard block and a loop block in a Wasm function.



Listing 2.3: Example of breaking a block and a loop in WebAssembly.

Each break instruction includes the depth of the enclosing block as an operand. This depth is used to identify the target block for the break instruction. For example, in the left-most part of the previously discussed listing, a break instruction with a depth of 1 would jump past two enclosing blocks. For the

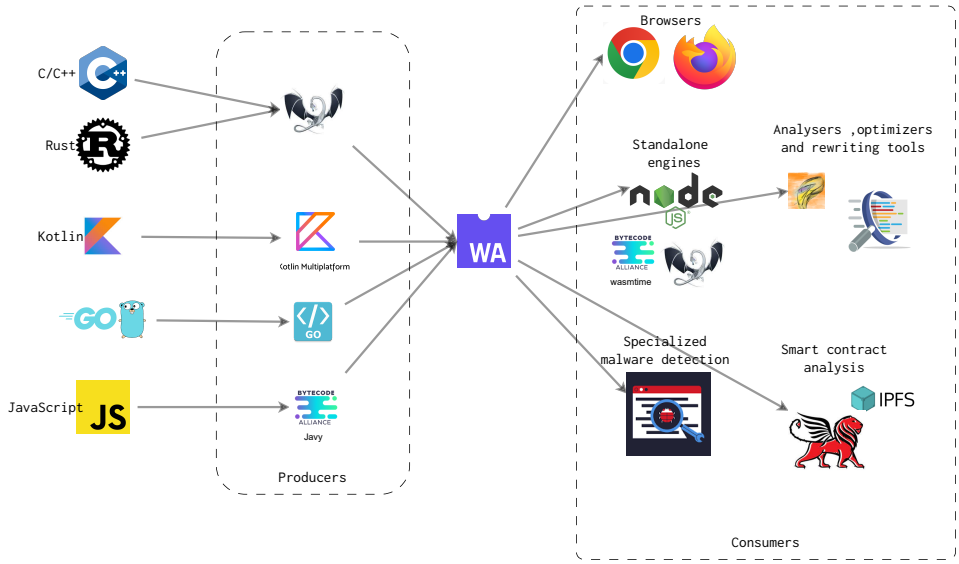


Figure 2.2: WebAssembly's ecosystem landscape separated into producers and consumers. For the sake of simplicity we do not include each tool mentioned in this chapter.

purposes of this dissertation, we introduce a specific term to describe a particular kind of break within loops:

Definition 1. *Break instructions within loops that effectively jump to the loop's beginning are termed* first-order breaks.

2.1.6 WebAssembly's Ecosystem

WebAssembly programs are designed for execution in host environments such as web browsers. Though the execution of a WebAssembly program might be considered its final lifecycle stage, the WebAssembly ecosystem is far from simplistic. It comprises multiple stakeholders and a rich array of tools that cater to various needs [10]. In Figure 2.2 we simplify the ecosystem landscape by separating it into producers, consumers and major stakeholders categories. In the subsequent text, we describe the WebAssembly ecosystem by separating it into stakeholders categories.

Producers , such as compilers, transform source code into WebAssembly binaries. For example, LLVM has offered WebAssembly as a backend option since its 7.1.0 release[3], supporting a diverse set of frontend languages like C/C++,

[3] `https://github.com/llvm/llvm-project/releases/tag/llvmorg-7.1.0`

Rust, Go, and AssemblyScript[4]. In parallel developments, the KMM framework[5] has incorporated WebAssembly as a compilation target, and the Javy approach[6] focuses on encapsulating JavaScript code within isolated WebAssembly binaries. This is achieved by porting both the engine and the source code into a secure WebAssembly environment. Blazor also enables the compilation of C code into WebAssembly binaries for browser execution[7]. Regardless of the source language or framework, the resulting WebAssembly binary functions similar to a traditional shared library, replete with code instructions, symbols, and exported functions.

Consumers encompass tools that undertake the tasks of validating, analyzing, optimizing, transpiling to machine code, and executing WebAssembly binaries, e.g. browser clients. In the text that follows, we dissect them into specific categories and their respective domains of application. Notice that, while some tools are designed for a specific domain, others are more general-purpose and might encapsulate more than one task in the WebAssembly ecosystem. For example, this is the case of browsers and standalone engines, which in one way or the other perform each one of the previous tasks.

Browser engines like V8[**?**], SpiderMonkey[**?**], and Chakra[**?**] are at the forefront of executing WebAssembly binaries in browser clients. These engines leverage Just-In-Time (JIT) compilers to convert WebAssembly into machine code. This translation is typically a straightforward one-to-one mapping, given that WebAssembly is already an optimized format closely aligned with machine code, as previously discussed in Subsection 2.1.3. For example, V8 employs quick, rudimentary optimizations such as constant folding and dead code removal[8].

Standalone engines WebAssembly's applicability has transcended browser environments, largely due to the WebAssembly System Interface (WASI)[5]. WASI aims to standardize the interaction between host environments and WebAssembly modules, thereby facilitating portability of both bytecode and binaries across diverse platforms. It outlines a POSIX-like Application Binary Interface (ABI), which includes functionalities for filesystem access, network sockets, clocks, and random number generation, among others. Standalone engines such as WASM3[**?**], wasmtime[**?**], WAVM[**?**] and Sledge [12] have emerged to support WebAssembly and WASI. Similarly, Singh and colleagues [13] proposed a virtual machine for Wasm in Arduino based devices.

TODO Disect them into, JIT compilers and interpreters, ending with SWAM.

TODO Add the verifiable standalone engine here

[4] A subset of the TypeScript language

[5] `https://kotlinlang.org/docs/wasm-overview.html`

[6] `https://github.com/bytecodealliance/javy`

[7] `https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor`

[8] This analysis was corroborated through discussions with the V8 development team and through empirical studies in one of our contributions[11]

TODO Locate WasmA [14]

Static and dynamic analysis, optimization and validation: Tools such as Wassail[15], Wasmati[16], and Wasp[17] utilize a range of strategies, including information flow control, code property graphs, and concolic execution, to identify vulnerabilities taking Wasm programs as input. Dynamic analysis counterparts like TaintAssembly[18], Wasabi[19], and Fuzzm[20] serve analogous functions. **TODO** Binanryen **TODO** `https://soft.vub.ac.be/Publications/2023/vub-tr-soft-23-11.pdf`, `https://dl.acm.org/doi/abs/10.1145/3510003.3510070?casa_token=hAlPiu9_oioAAAAA:DskwqdqM8c5-4e4W-8uU2CiENeyyOXWoi6BHytTiZUgyxojsYWXBsaS7ERFeqGNKfaYIKIN8pnQ1` **TODO** Wafl **TODO** Add veriwasm **TODO** Wasmfuzzer

Specialized Malware Detection In niche areas like cryptomalware detection, tools like MineSweeper[21], MinerRay[22], and MINOS[23] utilize static analysis techniques. Conversely, dynamic analysis is the forte of tools like SEISMIC[24], RAPID[25], and OUTGuard[26].

Smart Contract Analysis In the field of smart contracts, static analysis tools like EvalHunter[**?**], WANA[27], and EOSAFE[**?**] are employed to unearth vulnerabilities in WebAssembly-based contracts. Dynamic analysis tools in this sphere include EOSFuzzer[**?**] and wasai[28].

Obfuscation, and binary rewriting tools [29] [30] **TODO** WASMixer, Wobfuscator

While many existing tools purport to offer self-correctness evaluations and exhaustive test suites, the WebAssembly ecosystem is still in its early stages. To emphasize this, a 2021 study by Hilbig et al.[31] found a mere 8,000 unique WebAssembly binaries globally. This pales in comparison to mature ecosystems like JavaScript and Python, which offer 1.5 million and 1.7 million packages on npm[9] and PyPI[10], respectively. This limited pool of WebAssembly binaries presents a unique challenge for machine learning-based analysis tools, which require large datasets for effective training. This issue is explicitly addressed in one of the contributions of this dissertation[32]. Additionally, the scarcity of WebAssembly programs exacerbates the problem of software monoculture, increasing the likelihood of consuming a compromised WebAssembly program[**?**]. Besides, the current size of the WebAssembly ecosystem offers a small testing environment for evaluating the capabilities of consumer tools. This dissertation aims to address these issues by introducing a comprehensive suite of tools. These tools are crafted to not only bolster the security of WebAssembly programs through Software Diversification but also to intensify the testing rigor for both producers and consumers in the ecosystem.

[9] `https://www.npmjs.com/`
[10] `https://pypi.org/`

2.1.7 WebAssembly's Security

TODO Check the attack primitives defined by Lehmann. **TODO** Extend with more cases !

As we described, Wasm is deterministic and well-typed, follows a structured control flow and explicitly separates its linear memory model, global variables and the execution stack. This design is robust [33] and makes it easy for compilers and engines to sandbox the execution of Wasm binaries. Following the specification of Wasm for typing, memory, virtual stack and function calling, host environments should provide protection against data corruption, code injection, and return-oriented programming (ROP).

However, implementations in both browsers and standalone runtimes [34] are vulnerable. Genkin et al. demonstrated that Wasm could be used to exfiltrate data using cache timing-side channels [35]. Moreover, binaries itself can be vulnerable. The work of Lehmann et al. [9] proved that C/C++ source code vulnerabilities can propagate to Wasm such as overwriting constant data or manipulating the heap by overflowing the stack. Even though these vulnerabilities need a specific standard library implementation to be exploited, they make a call for better defenses for Wasm. Recently, Stiévenart and colleagues demonstrate that C/C++ source code vulnerabilities can be ported to Wasm [36]. Several proposals for extending Wasm in the current roadmap could address some existing vulnerabilities. For example, having multiple memories[11] could incorporate more than one memory, stack and global spaces, shrinking the attack surface. However, the implementation, adoption and settlement of the proposals are far from being a reality in all browser vendors[12].

2.2 Software diversification

2.2.2 Generating Software Diversification

2.2.3 Variants generation

2.2.4 Variants equivalence

TODO Automatic, SMT based **TODO** Take a look to Jackson thesis, we have a similar problem he faced with the superoptimization of NaCL **TODO** By design **TODO** Introduce the notion of rewriting rule by Sasnaukas. https://link.springer.com/chapter/10.1007/978-3-319-68063-7_13

[11] https://github.com/WebAssembly/multi-memory/blob/main/proposals/multi-memory/Overview.md

[12] https://webassembly.org/roadmap/

■ 2.3 Exploiting Software Diversification

- 2.3.2 Defensive Diversification
- 2.3.3 Offensive Diversification

03 AUTOMATIC SOFTWARE DIVERSIFICATION FOR WEBASSEMBLY

WebAssembly programs are produced ahead of time through a process that begins with the source code of the program and moves through a compiler, resulting in a WebAssembly program. Software Diversification can be achieved at any of these stages. Diversifying at the source code stage, however, is not practical due to the need of creating a distinct diversifier for each language compatible with WebAssembly. In contrast, focusing on the compiler stage presents a viable option, especially considering that 70% of WebAssembly binaries are created using LLVM-based compilers, as noted by Hilbig et al. [31]. Furthermore, implementing diversification at the WebAssembly program stage stands as the most generic strategy, applicable to any WebAssembly program in the wild. Therefore, this thesis focuses on the exploration of diversification strategies at the compiler and WebAssembly program stages, employing two main approaches: compiler-based and binary-based.

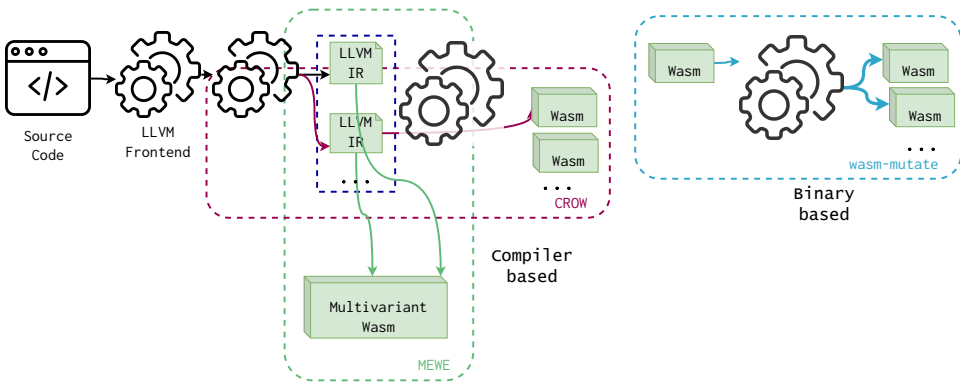


Figure 3.1: Approach landscape containing our three technical contributions: CROW squared in red, MEWE squared in green and WASM-MUTATE squared in blue. We annotate where our contributions, compiler-based and binary-based, stand in the landscape of generating WebAssembly programs.

Our compiler-based strategies are highlighted in red and green in Figure 3.1. This approach introduces a diversifier component in the LLVM pipeline, generating LLVM IR variants and producing artificial software diversity for Wasm. This strategy encompasses two tools: CROW [11], which creates Wasm program variants, and MEWE [37], which merges these variants to provide multivariant execution for Wasm. In contrast, the binary-based strategy, illustrated in blue in Figure 3.1, offers diversification for any WebAssembly program, removing the need for compiler tuning. WASM-MUTATE [38] generates a pool of WebAssembly program variants through rewriting rules upon an e-graph [39] data structure.

This dissertation contributes to the field of Software Diversification for WebAssembly, presenting three main technical contributions: CROW, MEWE, and WASM-MUTATE, dissected upon in the subsequent sections. Moreover, we compare our technical contributions between them. Furthermore, we outline the artifacts for our three technical contributions for the sake of open-research and reproducibility.

3.1 CROW: Code Randomization of WebAssembly

This section details CROW [11], represented as the red squared tooling in Figure 3.1. CROW is designed to produce semantically equivalent Wasm variants from the output of an LLVM front-end, utilizing a custom Wasm LLVM backend.

Figure 3.2 illustrates CROW's workflow in generating program variants, a process compound of two core stages: *exploration* and *combining*. During the *exploration* stage, CROW processes every instruction within each function of the LLVM input, creating a set of functionally equivalent code variants. This process ensures a rich pool of options for the subsequent stage. In the *combining* stage, these alternatives are assembled to form diverse LLVM IR variants, a task achieved through the exhaustive traversal of the power set of all potential combinations of code replacements. The final step involves the custom Wasm LLVM backend, which compiles the crafted LLVM IR variants into Wasm binaries.

3.1.2 Variants' generation

The primary component of CROW's exploration process is its code replacement generation strategy. The diversifier implemented in CROW is based on the proposed superdiversifier methodology of Jacob et al. [41]. A superoptimizer focuses on *searching* for a new program that is faster or smaller than the original code while preserving its functionality. The concept of superoptimizing a program dates back to 1987, with the seminal work of Massalin [42] which proposes an exhaustive exploration of the solution space. The search space is defined by choosing a subset of the machine's instruction set and generating combinations

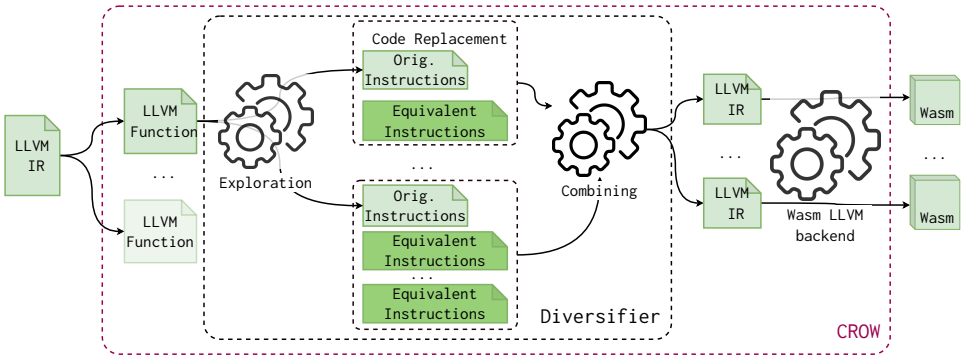


Figure 3.2: CROW components following the diagram in Figure 3.1. CROW takes LLVM IR to generate functionally equivalent code replacements. Then, CROW assembles program variants by combining them. Figure taken from [40].

of optimized programs, sorted by code size in ascending order. If any of these programs is found to perform the same function as the source program, the search halts. On the contrary, a superdiversifier keeps all intermediate search results despite their performance.

We build CROW upon an already existing superoptimizer for LLVM called Souper [43]. Yet, we modify it to keep all possible solutions in their searching algorithm. Souper builds a Data Flow Graph for each LLVM integer-returning instruction. Then, for each Data Flow Graph, Souper exhaustively builds all possible expressions from a subset of the LLVM IR language. Each syntactically correct expression in the search space is semantically checked versus the original with a theorem solver. Souper synthesizes the replacements in increasing size. Thus, the first found equivalent transformation is the optimal replacement result of the searching. CROW keeps more equivalent replacements during the searching by removing the halting criteria. Instead of the original halting conditions, CROW does not halt when it finds the first replacement. CROW continues the search until a timeout is reached or the replacements grow to a size larger than a predefined threshold.

Notice that the searching space increases exponentially with the size of the LLVM IR language subset. Thus, we prevent Souper from synthesizing instructions without correspondence in the Wasm backend. This decision reduces the searching space. For example, creating an expression having the `freeze` LLVM instructions will increase the searching space for instruction without a Wasm's opcode in the end. Moreover, we disable the majority of the pruning strategies of Souper for the sake of more program variants. For example, Souper prevents the generation of commutative operations during the searching. On the contrary, CROW still uses such transformation as a strategy to generate program variants.

The last stage involves the custom Wasm LLVM backend, which generates the Wasm programs. For it, we have the premise of removing all built-in optimizations in the LLVM backend that could reverse Wasm variants. We disable all optimizations in the Wasm backend that could reverse the CROW transformations.

3.1.3 Constant inferring

CROW, through using Souper adds a new transformation strategy that leads to more Wasm program variants, *constant inferring*. This means that Souper infers pieces of code as a single constant assignment. In particular, Souper focuses on variables that are used to control branches. After a *constant inferring*, the generated program is considerably different from the original program, being suitable for diversification.

Let us illustrate the case with an example. The Babbage problem code in Listing 3.1 is composed of a loop that stops when it discovers the smaller number that fits with the Babbage condition in Line 4.

```
1  int babbage() {
2      int current = 0,
3          square;
4      while ((square=current*current) %
           ↪ 1000000 != 269696) {
5          current++;
6      }
7      printf ("The number is %d\n",
           ↪ current);
8      return 0 ;
9  }
```

Listing 3.1: Babbage problem. Taken from [40].

```
1  int babbage() {
2      int current = 25264;
3
4
5
6
7
8
9      printf ("The number is %d\n", current);
10     return 0 ;
11 }
```

Listing 3.2: Constant inferring transformation over the original Babbage problem in Listing 3.1. Taken from [40].

In theory, this value can also be inferred by unrolling the loop the correct number of times with the LLVM toolchain. However, standard LLVM tools cannot unroll the **`while`**-loop because the loop count is too large. The original Souper deals with this case, generating the program in Listing 3.2. It infers the value of `current` in Line 2 such that the Babbage condition is reached. Therefore, the condition in the loop will always be false. Then, the loop is dead code and is removed in the final compilation. The new program in Listing 3.2 is remarkably smaller and faster than the original code. Therefore, it offers differences both statically and at runtime[1]

[1]Notice that for the sake of illustration, we show both codes in C language, this process inside CROW is performed directly in LLVM IR.

3.1.4 Combining replacements

When we retarget Souper, to create variants, we recombine all code replacements, including those for which a constant inferring was performed. This allows us to create variants that are also better than the original program in terms of size and performance. Most of the Artificial Software Diversification works generate variants that are as performant or iller than the original program. By using a superdiversifier, we could be able to generate variants that are better, in terms of performance, than the original program. This will give the option to developers to decide between performance and diversification without sacrificing the former.

On the other hand, when Souper finds a replacement, it is applied to all equal instructions in the original LLVM binary. In our implementation, we apply the transformations one by one. For example, if we find a replacement that is suitable for N difference places in the original program, we generate N different programs by applying the transformation in only one place at a time. Notice that this strategy provides a combinatorial explosion of program variants as soon as the number of replacements increases.

3.1.5 CROW instantiation

Let us illustrate how CROW works with the example code in Listing 3.3. The `f` function calculates the value of $2 * x + x$ where `x` is the input for the function. CROW compiles this source code and generates the intermediate LLVM bitcode in the left most part of Listing 3.4. CROW potentially finds two integer returning instructions to look for variants, as the right-most part of Listing 3.4 shows.

```
1  int f(int x) {
2      return 2 * x + x;
3  }
```

Listing 3.3: C function that calculates the quantity $2x + x$.

```
define i32 @f(i32) {
  %2 = mul nsw i32 %0,2
  %3 = add nsw i32 %0,%2

  ret i32 %3
}

define i32 @main() {
  %1 = tail call i32 @f(
       i32 10)
  ret i32 %1
}
```

```
Replacement candidates
      for code_1
%2 = mul nsw i32 %0,2

%2 = add nsw i32 %0,%0

%2 = shl nsw i32 %0, 1:i32
```

```
Replacement candidates for
      code_2
%3 = add nsw i32 %0,%2

%3 = mul nsw %0, 3:i32
```

Listing 3.4: LLVM's intermediate representation program, its extracted instructions and replacement candidates. Gray highlighted lines represent original code, green for code replacements.

```
%2 = mul nsw i32 %0,2
%3 = add nsw i32 %0,%2

%2 = add nsw i32 %0,%0
%3 = add nsw i32 %0,%2

%2 = shl nsw i32 %0, 1:i32
%3 = add nsw i32 %0,%2
```

```
%2 = mul nsw i32 %0,2
%3 = mul nsw %0, 3:i32

%2 = add nsw i32 %0,%0
%3 = mul nsw %0, 3:i32

%2 = shl nsw i32 %0, 1:i32
%3 = mul nsw %0, 3:i32
```

Listing 3.5: Candidate code replacements combination. Orange highlighted code illustrate replacement candidate overlapping.

CROW, detects `code_1` and `code_2` as the enclosing boxes in the left most part of Listing 3.4 shows. CROW synthesizes $2 + 1$ candidate code replacements for each code respectively as the green highlighted lines show in the right most parts of Listing 3.4. The baseline strategy of CROW is to generate variants out of all possible combinations of the candidate code replacements, *i.e.,* uses the power set of all candidate code replacements.

In the example, the power set is the cartesian product of the found candidate code replacements for each code block, including the original ones, as Listing 3.5 shows. The power set size results in 6 potential function variants. Yet, the generation stage would eventually generate 4 variants from the original program. CROW generated 4 statically different Wasm files, as Listing 3.6 illustrates. This gap between the potential and the actual number of variants is a consequence of the redundancy among the bitcode variants when composed into one. In other words, if the replaced code removes other code blocks, all possible combinations having it will be in the end the same program. In the example case, replacing `code_2` by `mul nsw %0, 3`, turns `code_1` into dead code, thus, later replacements generate the same program variants. The rightmost part of Listing 3.5 illustrates how for three different combinations, CROW produces the same variant. We call this phenomenon a *code replacement overlapping*.

```
func $f (param i32) (result i32)
   local.get 0
    i32.const 2
    i32.mul
    local.get 0
    i32.add

func $f (param i32) (result i32)
    local.get 0
    local.get 0
    i32.add
    local.get 0
    i32.add
```

```
func $f (param i32) (result i32)
    local.get 0
    i32.const 1
    i32.shl
    local.get 0
    i32.add

func $f (param i32) (result i32)
    local.get 0
    i32.const 3
    i32.mul
```

Listing 3.6: Wasm program variants generated from program Listing 3.3.

One might think that a reasonable heuristic could be implemented to avoid such overlapping cases. Instead, we have found it easier and faster to generate the variants with the combination of the replacement and check their uniqueness after the program variant is compiled. This prevents us from having an expensive checking for overlapping inside the CROW code. Still, this phenomenon calls for later optimizations in future works.

Contribution paper and artifact

CROW fully presented in Cabrera-Arteaga et al. "CROW: Code Randomization of WebAssembly" *Network and Distributed System Security Symposium, MADWeb* `https://doi.org/10.14722/madweb.2021.23004`.

CROW source code is available at `https://github.com/ASSERT-KTH/slumps`

3.2 MEWE: Multi-variant Execution for WebAssembly

This section describes MEWE [37]. MEWE synthesizes diversified function variants by using CROW. It then provides execution-path randomization in a Multivariant Execution (MVE). MEWE generates application-level multivariant binaries without changing the operating system or Wasm runtime. It creates an MVE by intermixing functions for which CROW generates variants, as illustrated by the green square in Figure 3.1. MEWE inlines function variants when appropriate, resulting in call stack diversification at runtime.

In Figure 3.3, we focus on MEWE, highlighted in green in Figure 3.1. MEWE takes the LLVM IR variants generated by CROW's diversifier. It then merges LLVM IR variants into a Wasm multivariant. In the figure, we highlight the two components of MEWE, *Multivariant Generation* and the *Mixer*. In the *Multivariant Generation* process, MEWE merges the LLVM IR variants created by CROW and creates an LLVM multivariant binary. The merging of the variants intermixes the calling of function variants, allowing the execution path randomization.

The Mixer augments the LLVM multivariant binary with a random generator. The random generator is needed to perform the execution-path randomization. Also, *The Mixer* fixes the entrypoint in the multivariant binary. Finally, using the same custom Wasm LLVM backend as CROW, MEWE generates a standalone multivariant Wasm binary. Once generated, the multivariant Wasm binary can be deployed to any Wasm engine.

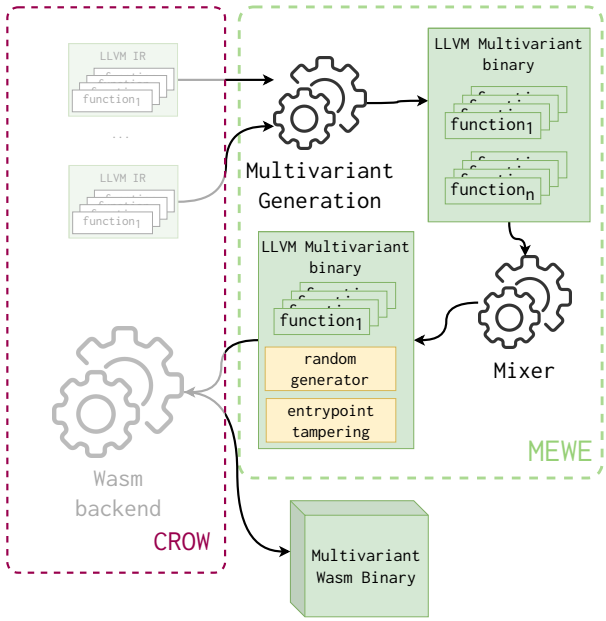


Figure 3.3: Overview of MEWE workflow. It takes as input an LLVM binary. It first generates a set of functionally equivalent variants for each function in the binary using CROW. Then, MEWE generates an LLVM multivariant binary composed of all the function variants. Finally, the Mixer includes the behavior in charge of selecting a variant when a function is invoked. Finally, the MEWE mixer composes the LLVM multivariant binary with a random number generation library and tampers the original application entrypoint. The final process produces a Wasm multivariant binary ready to be deployed. Figure partially taken from [40].

3.2.2 Multivariant generation

The key component of MEWE consists of combining the variants into a single binary. The goal is to support execution-path randomization at runtime. The core idea is to introduce one dispatcher function per original function with variants. A dispatcher function is a synthetic function in charge of choosing a variant at random when the original function is called. With the introduction of the dispatcher function, MEWE turns the original call graph into a multivariant call graph, defined as follows.

In Figure 3.4, we show the original static call graph for an original program (top of the figure), as well as the multivariant call graph generated with MEWE (bottom of the figure). The gray nodes represent function variants, the green nodes function dispatchers, and the yellow nodes are the original functions. The directed edges represent the possible calls. The original program includes three

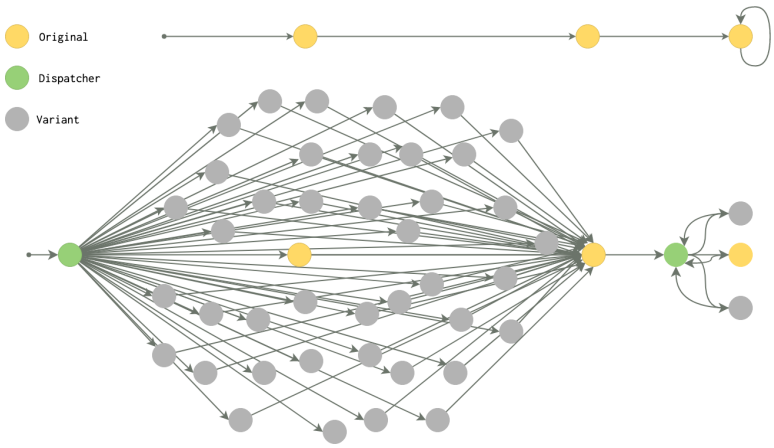


Figure 3.4: Example of two static call graphs. At the top, is the original call graph, and at the bottom, is the multivariant call graph, which includes nodes that represent function variants (in gray), dispatchers (in green), and original functions (in yellow). Figure taken from [40].

functions. MEWE generates 43 variants for the first function, none for the second, and three for the third. MEWE introduces two dispatcher nodes for the first and third functions. Each dispatcher is connected to the corresponding function variants to invoke one variant randomly at runtime.

```
define internal i32 @foo(i32 %0) {
    entry:
      %1 = call i32 @discriminate(i32 3)
      switch i32 %1, label %end [
        i32 0, label %case_43_
        i32 1, label %case_44_
      ]
    case_43_:
      %2 = call i32 @foo_43_(%0)
      ret i32 %2
    case_44_:
      %3 = <body of foo_44_ inlined>
      ret i32 %3
    end:
      %4 = call i32 @foo_original(%0)
      ret i32 %4
}
```

Listing 3.7: Dispatcher function embedded in the multivariant binary of the original function in the rightmost green node in Figure 3.4.

In Listing 3.7, we illustrate the LLVM construction for the function dispatcher corresponding to the right most green node of Figure 3.4. It first calls the random

generator, which returns a value used to invoke a specific function variant. We utilize a switch-case structure in the dispatchers to prevent indirect calls, which are vulnerable to speculative execution-based attacks [34]. The choice of a switch-case also avoids having multiple function definitions with the same signature, which could increase the attack surface in case the function signature is vulnerable [44]. This also allows MEWE to inline function variants inside the dispatcher instead of defining them again.

MEWE has four specific objectives: link the LLVM multivariant binary, inject a random generator, tamper the application's entrypoint, and merge all these components into a multivariant Wasm binary. We use the Rustc compiler[2] to orchestrate the mixing. For the random generator, we rely on WASI's specification [5] for the random behavior of the dispatchers. However, its exact implementation is dependent on the platform on which the binary is deployed. The Mixer component of MEWE creates a new entrypoint for the binary called *entrypoint tampering*. It wraps the dispatcher for the entrypoint variants as a new function for the final Wasm binary and is declared as the application entrypoint.

Contribution paper and artifact

MEWE is fully presented in Cabrera-Arteaga et al. "Multi-Variant Execution at the Edge" *Conference on Computer and Communications Security, MTD* `https://dl.acm.org/doi/abs/10.1145/3560828.3564007`

MEWE is also available as an open-source tool at `https://github.com/ASSERT-KTH/MEWE`

3.3 WASM-MUTATE: Fast and Effective Binary for WebAssembly

In this section, we introduce our third technical contribution, WASM-MUTATE [38], a tool that generates functionally equivalent variants of a WebAssembly program input. Leveraging rewriting rules and e-graphs [39] for diversification space traversals, WASM-MUTATE synthesizes program variants by altering parts of the original binary. In Figure 3.1, we highlight WASM-MUTATE as the blue squared tooling for a visual representation.

Figure 3.5 illustrates the workflow of WASM-MUTATE, which initiates with a WebAssembly binary as its input. The first step involves parsing this binary to create suitable abstractions, e.g. an intermediate representation. Subsequently, WASM-MUTATE utilizes predefined rewriting rules to construct an e-graph

[2] `https://doc.rust-lang.org/rustc/what-is-rustc.html`

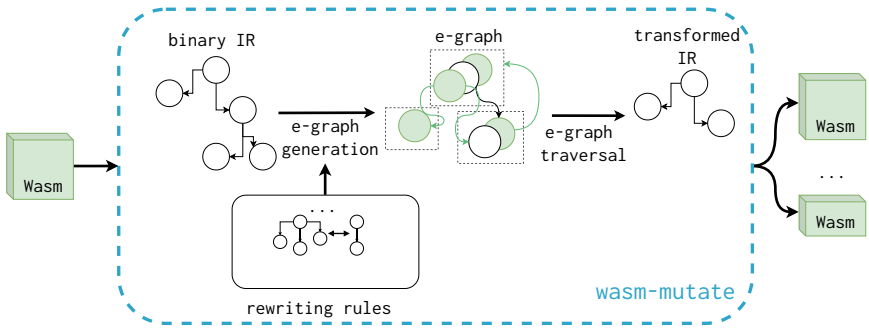


Figure 3.5: WASM-MUTATE high-level architecture. It generates semantically equivalent variants from a given WebAssembly binary input. Its central approach involves synthesizing these variants by substituting parts of the original binary using rewriting rules, boosted by diversification space traversals using e-graphs.

for the initial program, encapsulating all potential equivalent codes derived from the rewriting rules. Then, pieces of the original program are randomly substituted by the result of random e-graph traversals, resulting in a variant that maintains functional equivalence to the original binary. This assurance of semantic preservation is rooted in the inherent properties of the individual rewrite rules employed.

3.3.2 WebAssembly Rewriting Rules

WASM-MUTATE incorporates a total of 135 rewriting rules, organized into categories referred to as meta-rules. The rewriting rules are conceived based on the seminal work of Sasnauskas et al. [45], extended to include a predicate to enforce the conditions for replacement. Each rule is articulated as a tuple, represented as `(LHS, RHS, Cond)`, where: `LHS` identifies the segment of code targeted for replacement, `RHS` outlines the functionally equivalent substitute, and `Cond` defines the circumstances permitting the replacements. For example, in the case of WebAssembly binaries, the `Cond` predicate ensures that the replacement does not violate the type constraints. The following text details the seven meta-rules utilized in WASM-MUTATE.

Add type: WASM-MUTATE implements two rewrite rules for the Type Section of the input WebAssembly program. These rewriting rules create random function signatures, varying both, the number of parameters and the count of results. It ensures the consistency of the index of already defined types, even after the introduction of a new type. The following listing illustrates how WASM-MUTATE adds a new type definition based on this meta-rule.

```
LHS (module
  (type (;0;) (func (param i32) (result i64)))

---

RHS (module
  (type (;0;) (func (param i32) (result i64)))
  (type (;0;) (func (param i64 ... ) (result i32 ... )))
```

Add function: WASM-MUTATE adds new random functions by mutating the code, type, and function sections. This process begins with the creation of a random type signature, followed by the formulation of a random function body that simply returns the default value corresponding to the result type. An illustration of this transformation is provided in the subsequent example.

```
LHS (module
   (type (;0;) (func (param i32 f32) (result i64)))

---

RHS (module
   (type (;0;) (func (param T) (result t)))
   (func (;0;) (type 0) (param T) (result t)
     t.const 0)
```

Debloat: WASM-MUTATE randomly eliminates dead parts of the input Wasm program, targeting specific elements such as *functions, types, custom sections, imports, tables, memories, globals, data segments, and elements* that are verifiably unused. For instance, the removal of a memory declaration needs the absence of any memory access operations within the binary code. WASM-MUTATE incorporates distinct mutators for each element type to facilitate this process. The following example showcases a function removal using this meta-rule.

```
LHS (module (import "" "" (func ))))

---

RHS (module )

Cond The removed function is not called, it is not exported, and it is not
     in the binary table.
```

Edit custom sections: WASM-MUTATE randomly changes either the content or the name of the custom section, a process illustrated in the subsequent example.

```
LHS (module
...
      (@custom "CS42" "zzz..."
```

```
RHS (module
...
       (@custom "..." "...")
```

If swapping: In WebAssembly, the if-construction is a compound of two paths: the consequence and the alternative. The determination of which execution path to follow depends on the branching condition evaluated just before the `if` instruction. Specifically, a value greater than `0` at the top of the execution stack triggers the execution of the consequence code, while any other outcome initiates the alternative code. The *if swapping* rewriting rule interchanges the consequence and alternative codes within the if-construction, effectively reversing the original paths defined by the condition.

To facilitate the swapping of an if-construction in WebAssembly, WASM-MUTATE introduces a negation of the value situated at the top of the stack immediately preceding the `if` instruction. The methodology behind this rewriting is demonstrated in the following example.

```
LHS (module
    (func ...) (
 condition C
       (if A else B end)
    )
)
```

```
RHS (module
    (func ...) (
 condition C
i32.eqz
       (if B else A end)
    )
)
```

In this context, the consequence and alternative codes are labeled with the letters `A` and `B`, respectively, while the if-construction's condition is represented as `C`. To negate this condition, the `i32.eqz` instruction is incorporated into the RHS segment of the rewriting rule, functioning to compare the stack's top value with zero and, if true, pushing the value `1` onto the stack. In addition, WASM-MUTATE introduces a `nop` instruction to substitute for the absent code block, ensuring a seamless rewriting process.

Loop Unrolling: WASM-MUTATE randomly unrolls loops. To unroll a loop WASM-MUTATE first creates a new Wasm block, which contains a copy of its

body (unrolling) followed by the original loop. The copy of the loop body is itself a Wasm block. To maintain the original control flow functionality, the instructions inside the loop and their copied body need to be adjusted. To adjust the loop, WASM-MUTATE modifies the loop instructions that are first-order breaks, i.e., jumps that lead back to the loop's beginning and end (see section Subsection 2.1.4).

Inside the loop's body, there can be two types of first-order breaks: the first type which leads back to the loop's beginning, and the second type jumps, which leads to the loop's end. The second type is irrelevant for the unrolling process since the loop's end is not modified. To adjust first-order breaks, in the case of the copied body, they need to break the Wasm block that contains the loop body copy, making the execution of the program continue with the loop's original construction appended after it. In the case of the original loop, the first-order breaks need to interrupt the block that contains the loop body, making the execution of the program to continue as originally as the loop finishes. In concrete, their jumping indexes need to be incremented by one, going outside the loop-unrolling outer Wasm block. In the following example, we illustrate the unrolling of a loop.

```
LHS (module
    (func ...) (
        (loop   A br_if 0   B end)
  )
)
```

```
RHS (module
    (func ...) (
        (block
            (block A'   br_if 0   B'   br 1   end)
            (loop   A'   br_if 0   B'   end)
        end)
    )
)
```

In the LHS part of the rewriting rule, the loop showcases the first-order break. The loop concludes just before the `end` instruction if the break is not triggered. When WASM-MUTATE unrolls this loop, it undergoes a bifurcation of its instructions into two distinct groups, `A` and `B`. The RHS part of the illustration creates the two fresh Wasm blocks used for unrolling. Here, the outer block is a container for both the original and the duplicated loop body, while the inner entities, labeled `A'` and `B'`, embody adjustments to the jump directives originally found in groups `A` and `B`. Moreover, the conclusion of the unrolled loop body copy is marked by the insertion of an unconditional branch `br 1`. This strategic placement guarantees that, in the absence of a continuation in the loop body, the operation exits the scope.

Peephole: This meta-rule focuses on the rewriting of instruction sequences

found within function bodies, representing the lowest level of rewriting. In WASM-MUTATE, we have devised 125 rewriting rules specifically for this category. WASM-MUTATE is structured to ensure the determinism of the instructions selected for replacement. Therefore, any rewriting rule inside the Peephole meta-rule avoids instructions that might induce undefined behavior, e.g., function calls. Consequently, the scope of this meta-rule is confined to modifications in stack and memory operations, preserving the original functionality of the control frame labels.

The peephole category rewriting rules are meticulously designed and manually verified. An instance of a rewriting rule in this category can be appreciated below:

```
LHS (x)
```

```
RHS (x i32.or x)

Cond x is i32 type
```

The previous rewriting rule example implies that the `LHS` 'x' is to be replaced by an idempotent bitwise `i32.or` operation with itself, as soon as x, which can be any subexpression, leaves a value of type i32 in the execution stack.

3.3.3 Extending peephole meta-rules with custom operators

As illustrated in Figure 3.5, the initial step in the process involves parsing an input WebAssembly program, generating an intermediate representation. This step facilitates the transition of the WebAssembly program to the next stages of WASM-MUTATE. This representation extends the textual Wat format. We augment it with custom operator instructions to enhance the transformation capabilities of WASM-MUTATE.

Custom operator instructions form part of the lowest level of transformation we provide in WASM-MUTATE, the Peephole meta-rule. These custom operator instructions are designed to bolster WASM-MUTATE by utilizing well-established code diversification techniques through rewriting rules. WASM-MUTATE includes four custom operator instructions in its intermediate representation of Wasm programs. In the following text, we describe each one of them. We also show concrete rewriting rules in the Peephole meta-rule that use them.

container : it acts as a holder for multiple instructions, enabling transformations without altering the program's semantics. Below, we demonstrate a rewriting rule that leverages it to insert `nop` instructions into the any WebAssembly program place, a well-known low-level diversification strategy [**?**]:

```
LHS x

---

RHS (container (x nop))
```

The instruction `x` (it can be a complete subexpression), can be substituted with `container (x nop)`. Thus, when converting back the intermediate representation to Wasm, a `nop` opcode is appended to the original instruction.

`useglobal:` this operator is used in a rewriting rule that substitutes its operand with the setting and retrieval actions involving a newly created global variable. In the following listing, we illustrate such a rewriting rule.

```
LHS x

---

RHS (useglobal x)
```

This rewriting rule is meant to stress the managed memory through random access to global variables.

unfold: this operator, working with a constant numeric operand, statically generates two numbers whose sum equals the constant, followed by the addition of operations for these numbers.

```
LHS (i32.const x)

---

RHS (unfold x)
```

rand: this operator injects random constant numbers in any place of the program. One of the rewriting rules using this operator is shown below.

```
LHS x

---

RHS (container ( x drop (rand) ))
```

In this rewriting rule, a subexpression is replaced by a container for which operands are the subexpression itself and the pushing of a random value into the execution stack, to be removed after with a `drop` instruction.

In practice, custom operators are only part of the rewriting rules of WASM-MUTATE. This means that, when converting Wasm to the intermediate representation no custom operator is generated. When converting back to the WebAssembly binary format from the intermediate representation, custom instructions are meticulously handled to retain the original functionality of the WebAssembly program. For example, the `container` custom operator is removed while its operands are encoded back to Wasm in their corresponding opcodes.

3.3.4 E-graphs

We developed WASM-MUTATE leveraging e-graphs, a specific graph data structure for representing rewriting rules [46]. Within an e-graph, there exist two distinct node types: e-nodes and e-classes. The former encapsulates either an operator or an operand present in the rewriting rule, while the latter groups e-nodes into equivalence classes, essentially serving as a composite virtual node that contains a collection of e-nodes. Consequently, each e-class contains at least one e-node. e-nodes have edges delineating the operator-operand equivalence relations between e-classes.

In the context of WASM-MUTATE, the e-graph is constructed from a WebAssembly program. This entails the transformation of each distinct expression, operator, and operand into e-nodes. A primer e-graph is built from the original program. This initial e-graph is subsequently augmented with e-nodes and e-classes derived from each one of the rewriting rules.

Let us illustrate the e-graph construction with a program that consists of a single instruction that returns an integer constant, denoted as `i64.const 0`. Besides, assume a single rewriting rule defined as `(x, x i64.or x, x returns type i64)`. In Figure 3.6 we visualize how the e-graph is built out of the program and rewriting rule.

Building the e-graph begins with the incorporation of the solitary program instruction, `i64.const 0`, as an e-node ①. Then, we derive additional e-nodes from the rewriting rule ②. This involves introducing a fresh e-node labeled `i64.or` and establishing edges leading to the `x` e-node. Since there is no extra condition in which the operator-operand relation is valid, `x` represents any e-class in the e-graph. Thus, the equivalence by merging the two e-nodes is affirmed, thus forming a unified e-class ③. Finally, we successfully construct an e-graph that encapsulates the relationships and equivalences dictated by the initial program and the rewriting rule ④, setting the stage for further analyses and transformations based on this structured representation.

3.3.5 Random e-graph traversal for variants generation

Willsey et al. demonstrated the potential for high flexibility in extracting code fragments from e-graphs, a process that can be recursively orchestrated through a cost function applied to e-nodes and their respective operands. This methodology ensures the semantic equivalence of the derived code [39]. For instance, e-graphs solve the problem of providing the best code out of several optimization rules [**?**]. To extract the "optimal" code from an e-graph, one might commence the extraction at a specific e-node, subsequently selecting the AST with the minimal size from the available options within the corresponding e-class's operands. In omitting the cost function from the extraction strategy leads us to a significant property: *any path navigated through the e-graph yields a semantically equivalent code variant.*

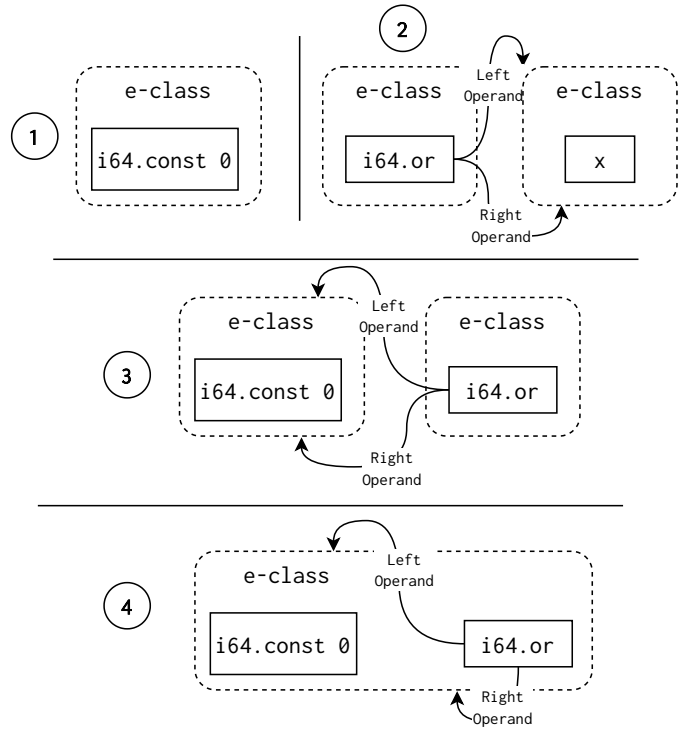


Figure 3.6: e-graph construction for idempotent bitwise-or rewriting rule and a single instruction Wasm program. Solid lines represent operand-operator relations, and dashed lines represent equivalent class inclusion.

The previously mentioned property is exploited in Figure 3.6, showcasing the feasibility of crafting an endless series of "or" operations. We exploit such property to generate diverse WebAssembly variants. We propose and implement an algorithm that facilitates the random traversal of an e-graph to yield semantically equivalent program variants, as detailed in Algorithm 1. This algorithm operates by taking an e-graph, an e-class node (starting with the root's e-class), and a parameter specifying the maximum extraction depth of the expression, to prevent infinite recursion. Within the algorithm, a random e-node is chosen from the e-class (as seen in lines 5 and 6), setting the stage for a recursive continuation with the offspring of the selected e-node (refer to line 8). Once the depth parameter reaches zero, the algorithm extracts the most concise expression available within the current e-class (line 3). Following this, the subexpressions are built (line 10) for each child node, culminating in the return of the complete expression (line 11).

Algorithm 1 e-graph traversal algorithm taken from [38].

```
1: procedure TRAVERSE(egraph, eclass, depth)
2:     if depth = 0 then
3:         return smallest_tree_from(egraph, eclass)
4:     else
5:         nodes ← egraph[eclass]
6:         node ← random_choice(nodes)
7:         expr ← (node, operands = [])
8:         for each child ∈ node.children do
9:             subexpr ← TRAVERSE(egraph, child, depth − 1)
10:            expr.operands ← expr.operands ∪ {subexpr}
11:        return expr
```

3.3.6 WASM-MUTATE instantiation

Let us illustrate how WASM-MUTATE generates variant programs by using the before enunciated algorithm. Here, we use Algorithm 1 with a maximum depth of 1. In Listing 3.8 a hypothetical original Wasm binary is illustrated. In this context, a potential user has set two pivotal rewriting rules: `(x, container (x nop),)` and `(x, x i32.add 0, x instanceof i32)`. The initial rule, which has been previously discussed in Subsection 3.3.3, grants the ability to append a `nop` instruction to any subexpression within the program. Conversely, the latter rule articulates the equivalence of augmenting any numeric value by zero.

```
(module
    (type (;0;) (func (param i32 f32) (result i64)))
    (func (;0;) (type 0) (param i32 f32) (result i64)
        i64.const 1)
)
```

Listing 3.8: Wasm function.

```
(module
    (type (;0;) (func (param i32 f32) (result i64)))
    (func (;0;) (type 0) (param i32 f32) (result i64)
        (i64.add (
            i64.const 0
            i64.const 1
            nop
        ))
    )
```

Listing 3.9: Random peephole mutation using egraph traversal for Listing 3.8 over e-graph Figure 3.7. The textual format is folded for better understanding.

Leveraging the code presented in Listing 3.8 alongside the defined rewriting

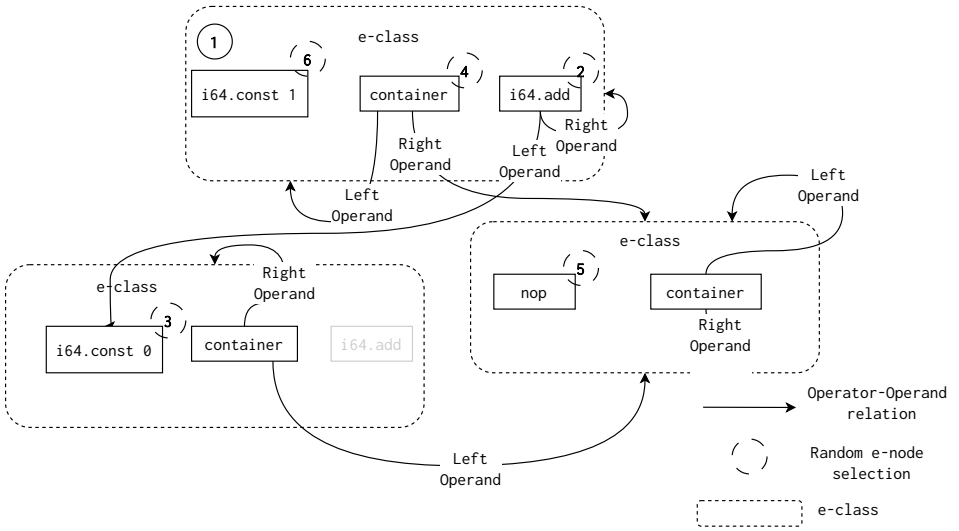


Figure 3.7: e-graph built for rewriting the first instruction of Listing 3.8.

rules, we build the e-graph, simplified in Figure 3.7. In the figure, we highlight various stages of Algorithm 1 in the context of the scenario previously described. The algorithm initiates at the e-class with the instruction `i64.const 1`, as seen in Listing 3.8. At ②, it randomly selects an equivalent node within the e-class, in this instance taking the `i64.add` node, resulting: `expr = i64.add l r`. As the traversal advances, it follows on the left operand of the previously chosen node, settling on the `i64.const 0` node within the same e-class ③. Then, the right operand of the `i64.add` node is choosen, selecting the `container` ④ operator yielding: `expr = i64.or (i64.const 0 container ( r nop ))`. The algorithm chooses the right operand of the `container` ⑤, which correlates to the initial instruction e-node highlighted in ⑥, culminating in the final expression: `expr = i64.or (i64.const 0 container( i64.const 1 nop)) i64.const 1`. As we proceed to the encoding phases, the `container` operator is ignored as a real Wasm instruction, finally resulting in the program in Listing 3.9.

Notice that, within the e-graph showcased in Figure 3.7, the container node maintains equivalence across all e-classes. Consequently, increasing the depth parameter in Algorithm 1 would potentially escalate the number of viable variants infinitely.

Contribution paper and artifact

WASM-MUTATE is fully presented in Cabrera-Arteaga et al. "WASM-MUTATE: Fast and Effective Binary Diversification for WebAssembly" `https://arxiv.org/pdf/2309.07638.pdf`.

WASM-MUTATE is available at `https://github.com/bytecodealliance/wasm-tools/tree/main/crates/wasm-mutate` as a contribution to the bytecodealliance organization [**?**].

3.4 Comparing CROW, MEWE, and WASM-MUTATE

In this section, we discuss the main differences between CROW, MEWE, and WASM-MUTATE. We discuss the main differences between CROW, MEWE, and WASM-MUTATE according to three main dimensions: 1) the technology and approach of each one, 2) the strength of the generated variants, and, 3) the security guarantees of the variants generated by each tool. We select these three dimensions because they lead the implementation of our tools.

3.4.2 Technology and approach

CROW is a compiler-based strategy, needing access to the source code or its LLVM IR representation to work. Its core is a Satisfiability Modulo Theories (SMT) solver, ensuring the functional equivalence of the generated variants. This approach lays the groundwork for a universal LLVM superdiversifier, potentially extending its applications and adaptability to other technologies. MEWE extends the capabilities of CROW, utilizing the same underlying technology to create program variants. It goes a step further by packaging the LLVM IR variants into a Wasm multivariant, providing MVE through execution path randomization.

On the other hand, WASM-MUTATE is a semi-automated, binary-based tool, centralizing its core around e-graph traversals. This approach facilitates the creation of a pool of WebAssembly program variants through the meticulous application of rewriting rules on an e-graph data structure. This method removes the need for compiler adjustments, offering compatibility with any existing WebAssembly binary. Moreover, it highlights how extending intermediate representations could establish a general framework for binary rewriting in WebAssembly.

3.4.3 Strength of the generated variants

CROW and MEWE use enumerative synthesis and verify semantic equivalence through SMT solvers. This approach not only has the potential to exceed

handcrafted optimizations but also ensures that the transformations are preserved. In other words, the transformations generated out of CROW and MEWE are virtually irreversible, even following compiler optimizations. This is particularly remarkable in the case of *constant inferring* transformations (see Subsection 3.1.3). While CROW and MEWE do not require any extra input but the program to diversify, the speed of variant generation is intrinsically linked to the SMT solvers' efficiency, known to be slow. Besides, their variants' generation capabilities are limited by the *overlapping* phenomenon discussed in Subsection 3.1.5.

On the other hand, WASM-MUTATE adopts a semi-automatic approach, requiring users to set the rewriting rules. Thus, the responsibility of ensuring functional equivalence is transferred to the rule creation process. This tool offers a significant advantage over CROW and MEWE as it permits transformations in any section of a Wasm program, not just the code section. Moreover, it leverages a virtually cost-free e-graph traversal process, avoiding, as a direct consequence, the *overlapping* issue seen in CROW and MEWE, as detailed in Subsection 3.1.5. In addition, since WASM-MUTATE operates at the binary level, it can modify functions incorporated by the WebAssembly producer itself. For example, this is the case of the *wasm32-wasi* architecture. While the original program might have a few lines of code, the underlying compiler might inject more functions to support the *wasm32-wasi* architecture. Thus, augmenting the diversification space available to WASM-MUTATE. Moreover, WASM-MUTATE outperforms CROW and MEWE capabilities in terms of the number of generated variants. Yet, the changes made by WASM-MUTATE might not be as preserved as the ones generated by CROW and MEWE. Thus, the variants generated by WASM-MUTATE might be more susceptible of being reversed, e.g. by further optimization passes.

Remarkably, CROW, MEWE, and WASM-MUTATE generate variants that potentially improve the original program's runtime performance, demystifying that software diversification inherently compromises performance.

3.4.4 Security guarantees

CROW and MEWE generate distinct and highly preserved code variants. This means that these variants, each with unique WebAssembly codes, maintain their distinctiveness even after JIT compilers translate them into machine codes (see Subsection 2.1.6). WASM-MUTATE, while offering slightly reduced preservation in its generated variants compared to CROW and MEWE, still maintains the same security guarantees excepting the multivariant cases. Its ability to produce a greater number of variants can offset this preservation shortfall. The preservation feature significantly reduces the impact of side-channel attacks that exploit specific machine code instructions, e.g., port contention [47].

Furthermore, CROW and MEWE enhance security against timing-based attacks by creating variants that exhibit a wide range of execution times.

This strategy is especially prominent in MEWE's approach, which develops multivariants functioning on randomizing execution paths, thereby thwarting attempts at timing-based inference attacks. Consequently, attackers might find it exceedingly difficult to identify a specific variant through time profiling of a MEWE multivariant (see Subsection 4.2.2 for the use case impact). Adding another layer benefit, the integration of diverse variants into multivariants can potentially disrupt dynamic analysis tools such as symbolic executors [29]. Concretely, different control flows through a random discriminator, exponentially increase the number of possible execution paths, making multivariant binaries virtually unexplorable.

An advantage of WASM-MUTATE, compared to CROW and MEWE, is its capacity to transform non-code sections without impacting the runtime behavior of the original variant, a strategy that effectively shields against static binary analysis, including malware detection based on signature sets [32]. For instance, it can modify the type section of a WebAssembly program, a section typically utilized only for function signature validation during compilation and validation processes by the host engine. This thwarts compiler identification techniques, such as fingerprinting. Besides, it can be used for masquerading as a different compilation source. Thus, reducing the fingerprinting surface available to attackers.

CROW, MEWE, and WASM-MUTATE can alter the original program structure, either by eliminating dead code or by introducing additional elements. From a static perspective, such alterations serve to reduce potential attack surfaces, thereby impeding signature-based identification. Yet, modifying the layout of a WebAssembly program inherently affects its managed memory during runtime, a segment not overseen by the WebAssembly program itself (see section **??** for a detailed discussion on unmanaged memory). This aspect is especially important for CROW and MEWE, given that they do not directly address the WebAssembly memory model. Significantly, CROW and MEWE considerably alter the managed memory by modifying the layout of the WebAssembly program. For example, the *constant inferring* transformations significantly alter the layout of program variants, affecting unmanaged memory elements such as the returning address of a function.

Furthermore, WASM-MUTATE not only affects managed memory through changes in the WebAssembly program layout. It also adds rewriting rules to transform unmanaged memory instructions, e.g. the rewriting rule involving the `useglobal` custom operator previously discussed in Subsection 3.3.3. Memory alterations, either to the unmanaged or managed memories, have substantial security implications. For instance, they can counteract attacks by eliminating potential jump points that facilitate malicious activities within the binary, a preventive measure highlighted by Narayan et al. [48].

3.5 Conclusions

In this chapter, we discuss the technical specifics underlying our primary technical contributions. We elucidate the mechanisms through which CROW generates program variants. Following this, we outline the conceptual framework for a universal LLVM superdiversifier, laying a foundation for broader applicability and versatility. Subsequently, we discuss MEWE, offering a detailed examination of its role in forging MVE for WebAssembly. We also explore the details of WASM-MUTATE, highlighting its pioneering utilization of an e-graph traversal algorithm to spawn Wasm program variants. Remarkably, we undertake a comparative analysis of the three tools, delineating their respective benefits and limitations, alongside the potential security assurances they provide upon the program variants derived from them.

In **??**, we present four use cases that support the exploitation of these tools. **??** serves to bridge theory with practice, showcasing the tangible impacts and benefits realized through the deployment of CROW, MEWE, and WASM-MUTATE.

04 EXPLOITING SOFTWARE DIVERSIFICATION FOR WEBASSEMBLY

4.1 Offensive Software Diversification

4.1.2 **Use case 1:** Automatic testing and fuzzing of WebAssembly consumers

TODO We explain the CVE. Make the explanation around "indirect memory diversification"

4.1.3 **Use case 2:** WebAssembly malware evasion

TODO The malware evasion paper

4.2 Defensive Software Diversification

4.2.2 **Use case 3:** Multivariant execution at the Edge

TODO Disturbing of execution time. Go around the web timing attacks. `https://arxiv.org/pdf/2210.10523.pdf` Attack model for MEWE.

4.2.3 **Use case 4:** Speculative Side-channel protection

In concrete, distributing the unmodified binary to 100 machines would, essentially, creates 100 homogeneously vulnerable machines. However, let us illustrate the case with a different approach: each time the binary is replicated onto a different machine, we distribute a unique variant instead of the original binary. If we disseminate a unique variant, with X stacked transformations, to each machine, every system would run a distinct Wasm binary. Based on our findings, even when some binaries are still vulnerable, we can confidently say that if 100 variants of a vulnerable program, each furnished with X stacked transformations, are distributed, the impact of any potential attack is considerably mitigated. While it's true that some variants may retain their original vulnerabilities, not all

of them do. This significantly enhances overall security. Further reinforcing this point, let's consider the case of btb_leakage. In this scenario, a suite of 100 variants, each featuring at least 200 stacked transformations, ensures full protection against potential threats, effectively securing the entire infrastructure. Moreover, considering the results for the ret2spec attack, this property holds for the whole population of generated variants, despite the number of stacked transformations. Therefore, WASM-MUTATE as a software diversification tool, is a peemptive solution to potential attacks.

TODO Go around the last paper

05 CONCLUSIONS AND FUTURE WORK

[1] A. Haas, A. Rossberg, D. L. Schuff, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. F. Bastien, "Bringing the web up to speed with webassembly," *PLDI*, 2017.

[2] P. Mendki, "Evaluating webassembly enabled serverless approach for edge computing," in *2020 IEEE Cloud Summit*, pp. 161–166, 2020.

[3] M. Jacobsson and J. Wåhslén, "Virtual machine execution for wearables based on webassembly," in *EAI International Conference on Body Area Networks*, pp. 381–389, Springer, Cham, 2018.

[4] Bytecode Alliance , "Bytecode Alliance." `https://bytecodealliance.org/`, 2019.

[5] "Webassembly system interface." `https://github.com/WebAssembly/WASI`, 2021.

[6] D. Bryant, "Webassembly outside the browser: A new foundation for pervasive computing," in *Proc. of ICWE 2020*, pp. 9–12, 2020.

[7] B. Spies and M. Mock, "An evaluation of webassembly in non-web environments," in *2021 XLVII Latin American Computing Conference (CLEI)*, pp. 1–10, 2021.

[8] E. Wen and G. Weber, "Wasmachine: Bring iot up to speed with a webassembly os," in *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pp. 1–4, IEEE, 2020.

[9] D. Lehmann, J. Kinder, and M. Pradel, "Everything old is new again: Binary security of webassembly," in *29th USENIX Security Symposium (USENIX Security 20)*, USENIX Association, Aug. 2020.

[10] M. Kim, H. Jang, and Y. Shin, "Avengers, assemble! survey of webassembly security solutions," in *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, pp. 543–553, 2022.

[11] J. Cabrera Arteaga, O. Floros, O. Vera Perez, B. Baudry, and M. Monperrus, "Crow: code diversification for webassembly," in *MADWeb, NDSS 2021*, 2021.

[12] P. K. Gadepalli, S. McBride, G. Peach, L. Cherkasova, and G. Parmer, "Sledge: A serverless-first, light-weight wasm runtime for the edge," in *Proceedings of the 21st International Middleware Conference*, p. 265–279, 2020.

[13] R. Gurdeep Singh and C. Scholliers, "Warduino: A dynamic webassembly virtual machine for programming microcontrollers," in *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming*

Languages and Runtimes, MPLR 2019, (New York, NY, USA), pp. 27–36, ACM, 2019.

[14] F. Breitfelder, T. Roth, L. Baumgärtner, and M. Mezini, "Wasma: A static webassembly analysis framework for everyone," in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 753–757, 2023.

[15] Q. Stiévenart and C. De Roover, "Wassail: a webassembly static analysis library," in *Fifth International Workshop on Programming Technology for the Future Web*, 2021.

[16] T. Brito, P. Lopes, N. Santos, and J. F. Santos, "Wasmati: An efficient static vulnerability scanner for webassembly," *Computers & Security*, vol. 118, p. 102745, 2022.

[17] F. Marques, J. Fragoso Santos, N. Santos, and P. Adão, "Concolic execution for webassembly (artifact)," Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[18] W. Fu, R. Lin, and D. Inge, "Taintassembly: Taint-based information flow control tracking for webassembly," *arXiv preprint arXiv:1802.01050*, 2018.

[19] D. Lehmann and M. Pradel, "Wasabi: A framework for dynamically analyzing webassembly," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 1045–1058, 2019.

[20] D. Lehmann, M. T. Torp, and M. Pradel, "Fuzzm: Finding memory bugs through binary-only instrumentation and fuzzing of webassembly," *arXiv preprint arXiv:2110.15433*, 2021.

[21] R. K. Konoth, E. Vineti, V. Moonsamy, M. Lindorfer, C. Kruegel, H. Bos, and G. Vigna, "Minesweeper: An in-depth look into drive-by cryptocurrency mining and its defense," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1714–1730, 2018.

[22] A. Romano, Y. Zheng, and W. Wang, "Minerray: Semantics-aware analysis for ever-evolving cryptojacking detection," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1129–1140, 2020.

[23] F. N. Naseem, A. Aris, L. Babun, E. Tekiner, and A. S. Uluagac, "Minos: A lightweight real-time cryptojacking detection system.," in *NDSS*, 2021.

[24] W. Wang, B. Ferrell, X. Xu, K. W. Hamlen, and S. Hao, "Seismic: Secure in-lined script monitors for interrupting cryptojacks," in *Computer Security: 23rd European Symposium on Research in Computer Security, ESORICS 2018,*

Barcelona, Spain, September 3-7, 2018, Proceedings, Part II 23, pp. 122–142, Springer, 2018.

[25] J. D. P. Rodriguez and J. Posegga, "Rapid: Resource and api-based detection against in-browser miners," in *Proceedings of the 34th Annual Computer Security Applications Conference*, pp. 313–326, 2018.

[26] A. Kharraz, Z. Ma, P. Murley, C. Lever, J. Mason, A. Miller, N. Borisov, M. Antonakakis, and M. Bailey, "Outguard: Detecting in-browser covert cryptocurrency mining in the wild," in *The World Wide Web Conference*, pp. 840–852, 2019.

[27] D. Wang, B. Jiang, and W. Chan, "Wana: Symbolic execution of wasm bytecode for cross-platform smart contract vulnerability detection," *arXiv preprint arXiv:2007.15510*, 2020.

[28] W. Chen, Z. Sun, H. Wang, X. Luo, H. Cai, and L. Wu, "Wasai: uncovering vulnerabilities in wasm smart contracts," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 703–715, 2022.

[29] S. Cao, N. He, Y. Guo, and H. Wang, "WASMixer: Binary Obfuscation for WebAssembly," *arXiv e-prints*, p. arXiv:2308.03123, Aug. 2023.

[30] A. Romano, D. Lehmann, M. Pradel, and W. Wang, "Wobfuscator: Obfuscating javascript malware via opportunistic translation to webassembly," in *2022 2022 IEEE Symposium on Security and Privacy (SP) (SP)*, (Los Alamitos, CA, USA), pp. 1101–1116, IEEE Computer Society, may 2022.

[31] A. Hilbig, D. Lehmann, and M. Pradel, "An empirical study of real-world webassembly binaries: Security, languages, use cases," *Proceedings of the Web Conference 2021*, 2021.

[32] J. Cabrera-Arteaga, M. Monperrus, T. Toady, and B. Baudry, "Webassembly diversification for malware evasion," *Computers & Security*, vol. 131, p. 103296, 2023.

[33] D. Chen and W3C group, "WebAssembly documentation: Security." `https://webassembly.org/docs/security/`, 2020. Accessed: 18 June 2020.

[34] S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. Tullsen, *et al.*, "Swivel: Hardening webassembly against spectre," in *USENIX Security Symposium*, 2021.

[35] D. Genkin, L. Pachmanov, E. Tromer, and Y. Yarom, "Drive-by key-extraction cache attacks from portable code," *IACR Cryptol. ePrint Arch.*, vol. 2018, p. 119, 2018.

[36] Q. Stiévenart, C. De Roover, and M. Ghafari, "Security risks of porting c programs to webassembly," in *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, SAC '22, (New York, NY, USA), p. 1713–1722, Association for Computing Machinery, 2022.

[37] J. Cabrera Arteaga, P. Laperdrix, M. Monperrus, and B. Baudry, "Multi-Variant Execution at the Edge," *arXiv e-prints*, p. arXiv:2108.08125, Aug. 2021.

[38] J. Cabrera-Arteaga, N. Fitzgerald, M. Monperrus, and B. Baudry, "WASM-MUTATE: Fast and Effective Binary Diversification for WebAssembly," *arXiv e-prints*, p. arXiv:2309.07638, Sept. 2023.

[39] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha, "Egg: Fast and extensible equality saturation," *Proc. ACM Program. Lang.*, vol. 5, jan 2021.

[40] J. Cabrera Arteaga, "Artificial software diversification for webassembly," 2022. QC 20220909.

[41] M. Jacob, M. H. Jakubowski, P. Naldurg, C. W. N. Saw, and R. Venkatesan, "The superdiversifier: Peephole individualization for software protection," in *International Workshop on Security*, pp. 100–120, Springer, 2008.

[42] M. Henry, "Superoptimizer: a look at the smallest program," *ACM SIGARCH Computer Architecture News*, vol. 15, pp. 122–126, Nov 1987.

[43] R. Sasnauskas, Y. Chen, P. Collingbourne, J. Ketema, G. Lup, J. Taneja, and J. Regehr, "Souper: A Synthesizing Superoptimizer," *arXiv preprint 1711.04422*, 2017.

[44] E. Johnson, D. Thien, Y. Alhessi, S. Narayan, F. Brown, S. Lerner, T. McMullen, S. Savage, and D. Stefan, "Sfi safety for native-compiled wasm," *NDSS. Internet Society*, 2021.

[45] R. Sasnauskas, Y. Chen, P. Collingbourne, J. Ketema, G. Lup, J. Taneja, and J. Regehr, "Souper: A Synthesizing Superoptimizer," *arXiv e-prints*, p. arXiv:1711.04422, Nov. 2017.

[46] D. Cao, R. Kunkel, C. Nandi, M. Willsey, Z. Tatlock, and N. Polikarpova, "Babble: Learning better abstractions with e-graphs and anti-unification," *Proc. ACM Program. Lang.*, vol. 7, jan 2023.

[47] T. Rokicki, C. Maurice, M. Botvinnik, and Y. Oren, "Port contention goes portable: Port contention side channels in web browsers," in *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '22, (New York, NY, USA), p. 1182–1194, Association for Computing Machinery, 2022.

[48] S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. Tullsen, and D. Stefan, "Swivel: Hardening WebAssembly against spectre," in *30th USENIX Security Symposium (USENIX Security 21)*, pp. 1433–1450, USENIX Association, Aug. 2021.

Part II

Included papers

SUPEROPTIMIZATION OF WEBASSEMBLY BYTECODE

Javier Cabrera-Arteaga, Shrinish Donde, Jian Gu, Orestis Floros, Lucas Satabin, Benoit Baudry, Martin Monperrus
Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming (Programming 2021), MoreVMs

https://doi.org/10.1145/3397537.3397567

Superoptimization of WebAssembly Bytecode

Javier Cabrera Arteaga
KTH
Sweden
javierca@kth.se

Shrinish Donde
KTH
Sweden
shrinish@kth.se

Jian Gu
KTH
Sweden
jiagu@kth.se

Orestis Floros
KTH
Sweden
forestis@kth.se

Lucas Satabin
Mobimeo
Germany
lucas.satabin@gnieh.org

Benoit Baudry
KTH
Sweden
baudry@kth.se

Martin Monperrus
KTH
Sweden
martin.monperrus@csc.kth.se

ABSTRACT

Motivated by the fast adoption of WebAssembly, we propose the first functional pipeline to support the superoptimization of WebAssembly bytecode. Our pipeline works over LLVM and Souper. We evaluate our superoptimization pipeline with 12 programs from the Rosetta code project. Our pipeline improves the code section size of 8 out of 12 programs. We discuss the challenges faced in superoptimization of WebAssembly with two case studies.

CCS CONCEPTS

• **Software and its engineering → Source code generation**; *Retargetable compilers*; Software implementation planning.

KEYWORDS

superoptimization, webassembly, web, optimization, lllvm

ACM Reference Format:
Javier Cabrera Arteaga, Shrinish Donde, Jian Gu, Orestis Floros, Lucas Satabin, Benoit Baudry, and Martin Monperrus. 2020. Superoptimization of WebAssembly Bytecode. In *Companion Proceedings of the 4th International Conference on the Art, Science, and Engineering of Programming (<Programming'20> Companion), March 23–26, 2020, Porto, Portugal.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3397537.3397567

1 INTRODUCTION

After HTML, CSS, and JavaScript, WebAssembly (WASM) has become the fourth standard language for web development [7]. This new language has been designed to be fast, platform-independent, and experiments have shown that WebAssembly can have an overhead as low as 10% compared to native code [11]. Notably, WebAssembly is developed as a collaboration between vendors and has been supported in all major browsers since 2017.

The state-of-art compilation frameworks for WASM are Emscripten and LLVM [5, 6], they generate WASM bytecode from high-level languages (e.g. C, C++, Rust). These frameworks can apply a sequence of optimization passes to deliver smaller and faster binaries. In the web context, having smaller binaries is important, because they are delivered to the clients over the network, hence smaller binaries means reduced latency and page load time. Having smaller WASM binaries to reduce the web experience is the core motivation of this paper.

To reach this goal, we propose to use superoptimization. Superoptimization consists of synthesizing code replacements in order to further improve binaries, typically in a way better than the best optimized output from standard compilers [4, 15]. Given a program, superoptimization searches for alternate and semantically equivalent programs with fewer instructions [12]. In this paper, we consider the superoptimization problem stated as finding an equivalent WebAssembly binary such that the size of the binary code is reduced compared to the default one.

This paper presents a study on the feasibility of superoptimization of WebAssembly bytecode. We have designed a pipeline for WASM superoptimization, done by tailoring and integrating open-source tools. Our work is evaluated by building a benchmark of 12 programs and applying superoptimization on them. The pipeline achieves a median size reduction of 0.33% in the total number of WASM instructions.

To summarize, our contributions are:

- The design and implementation of a functional pipeline for the superoptimization of WASM.
- Original experimental results on superoptimizing 12 C programs from the Rosetta Code corpus.

2 BACKGROUND

2.1 WebAssembly

WebAssembly is a binary instruction format for a stack-based virtual machine. As described in the WebAssembly Core Specification [7], WebAssembly is a portable, low-level code format designed for efficient execution and compact representation. WebAssembly has been first announced publicly in 2015. Since 2017, it has been implemented by four major web browsers (Chrome, Edge, Firefox, and Safari). A paper by Haas et al. [11] formalizes the language and its type system, and explains the design rationale.

The main goal of WebAssembly is to enable high performance applications on the web. WebAssembly can run as a standalone VM or in other environments such as Arduino [10]. It is independent of any specific hardware or languages and can be compiled for

<Programming'20> Companion, March 23–26, 2020, Porto, Portugal

ACM ISBN 978-1-4503-7507-8/20/03…$15.00
https://doi.org/10.1145/3397537.3397567

modern architectures or devices, from a wide variety of high-level languages. In addition, WebAssembly introduces a memory-safe, sand-boxed execution environment to prevent common security issues, such as data corruption and security breaches.

Since version 8, the LLVM compiler framework supports the WebAssembly compilation target by default [6]. This means that all languages that have an LLVM front end can be directly compiled to WebAssembly. Binaryen [14], a compiler and toolchain infrastructure library for WebAssembly, supports compilation to WebAssembly as well. Once compiled, WASM programs can run within a web browser or in a standalone runtime [10].

2.2 Superoptimization

Given an input program, code superoptimization focuses on *searching* for a new program variant which is faster or smaller than the original code, while preserving its correctness [2]. The concept of superoptimizing a program dates back to 1987, with the seminal work of Massalin [12] which proposes an exhaustive exploration of the solution space. The search space is defined by choosing a subset of the machine's instruction set and generating combinations of optimized programs, sorted by length in ascending order. If any of these programs are found to perform the same function as the source program, the search halts. However, for larger instruction sets, the exhaustive exploration approach becomes virtually impossible. Because of this, the paper proposes a pruning method over the search space and a fast probabilistic test to check programs equivalence.

State of the art superoptimizers such as STOKE [16] and Souper [15] make modifications to the code and generate code rewrites. A cost function evaluates the correctness and performance of the rewrites. Correctness is generally estimated by running the code against test cases (either provided by the user or generated automatically, e.g. symbolic evaluation on both original and replacement code).

2.3 Souper

Souper is a superoptimizer for LLVM [15]. It enumerates a set of several optimization candidates to be replaced. An example of such a replacement is the following, replacing two instructions by a constant value:

```
%0:i32 = var (range=[1,0))
%1:i1 = ne 0:i32, %0
```

```
cand %1 1:i1
```

In this case, Souper finds the replacement for the variable %1 as a constant value (in the bottom part of the listing) instead of the two instructions above.

Souper is based on a Satisfiability Modulo Theories (SMT) solver. SMT solvers are useful for both verification and synthesis of programs [8]. With the emergence of fast and reliable solvers, program alternatives can be efficiently checked, replacing the probabilistic test of Massalin [12] as mentioned in subsection 2.2.

In the code to be optimized, Souper refers to the optimization candidates as *left-hand side* (LHS). Each LHS is a fragment of code that returns an integer and is a target for optimization. Two different

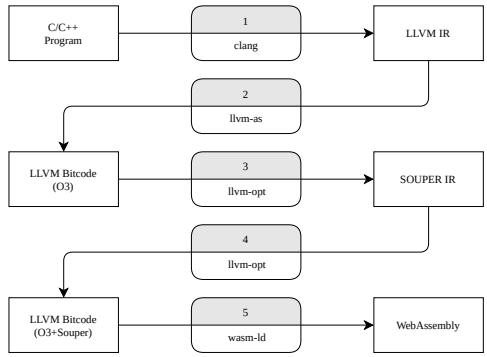


Figure 1: Superoptimization pipeline for WebAssembly based on Souper.

LHS candidates may overlap. For each candidate, Souper tries to find a *right-hand side* (RHS), which is a fragment of code that is combined with the LHS to generate a replacement. In the original paper's benchmarks [15], Souper optimization passes were found to further improve the top level compiler optimizations (-O3 for *clang*, for example) for some programs.

Souper is a platform-independent superoptimizer. The cost function is evaluated on an intermediate representation and not on the code generated for the final platform. Thus, the tool may miss optimizations that make sense for the target instruction set.

3 WASM SUPEROPTIMIZATION PIPELINE

The key contribution of our work is a superoptimization pipeline for WebAssembly. We faced two challenges while developing this pipeline: the need for a correct WASM generator, and the usage of a full-fledged superoptimizer. The combination of the LLVM WebAssembly backend and Souper provides the solution to tackle both challenges.

3.1 Steps

Our pipeline is a tool designed to output a superoptimized WebAssembly binary file for a given C/C++ program that can be compiled to WASM. With our pipeline, users write a high level source program and get a superoptimized WebAssembly version.

The pipeline (illustrated in Figure 1) first converts a high-level source language (e.g. C/C++) to the LLVM intermediate representation (LLVM IR) using the Clang compiler (Step 1). We use the code generation options in clang in particular the -O3 level of optimization which enables aggressive optimizations. In this step, we make use of the LLVM compilation target for WebAssembly 'wasm32-unknown-unknown'. This flag can be read as follows: wasm32 means that we target the 32 bits address space in WebAssembly; the second and third options set the compilation to any machine and performs inline optimizations with no specific strategy. LLVM IR is emitted as output.

Secondly, we use the LLVM assembler tool (llvm-as) to convert the generated LLVM IR to the LLVM bitcode file (Step 2). This LLVM assembler reads the file containing LLVM IR language, translates it to LLVM bitcode, and writes the result into a file. Thus, we make use of the optimizations from clang and the LLVM support for WebAssembly before applying superoptimization to the generated code.

Next, we use Souper, discussed in subsection 2.3, to add further superoptimization passes. Step 3 generates a set of optimized candidates, where a candidate is a code fragment that can be optimized by Souper. From this, Souper carries out a search to get shorter instruction sequences and uses an SMT solver to test the semantic equivalence between the original code snippet and the optimized one [15].

Step 4 produces a superoptimized LLVM bitcode file. The **opt** command is the LLVM analyzer that is shipped with recent LLVM versions. The purpose of the **opt** tool is to provide the capability of adding third party optimizations (plugins) to LLVM. It takes LLVM source files and the optimization library as inputs, runs the specified optimizations and outputs the optimized file or the analysis results. Souper is integrated as a specific pass for LLVM **opt**.

The last step of our pipeline consists of compiling the generated superoptimized LLVM bitcode file to a WASM program (Step 5). This final conversion is supported by the WebAssembly linker (wasm-ld) from the LLD project [13]. wasm-ld receives the object format (bitcode) that LLVM produces when run with the 'wasm32-unknown-unknown' target and produces WASM bytecode.

To our knowledge, this is the first successful integration of those tools into a working pipeline for superoptimizing WebAssembly code.

3.2 Insights

We note that Souper has been primarily designed with the LLVM IR in mind and requires a well-formed SSA representation of the program under superoptimization. The biggest challenge with WebAssembly is that there no complete transformation from WASM to SSA. In our pipeline, we work around this by assuming we have access to source code, this alternative path may be valid for plugging other binary format into Souper.

4 EXPERIMENTS

To study the effects and feasibility of applying superoptimization to WASM code, we run the superoptimization pipeline on a benchmark of programs.

The benchmark is based on the Rosetta Code corpus[1]. We have selected 12 C language programs that compile to WASM. Our selection of the programs is based on the following criteria:

(1) The programs can be successfully compiled to LLVM IR.
(2) They are diverse in terms of application domain.
(3) The programs are small to medium sized: between 15 and 200 lines of C code each.
(4) They have no dependencies to external libraries.

The code of each program is available as part of our experimental package[2].

[1] http://rosettacode.org
[2] https://github.com/KTH/slumps/tree/master/utils/pipeline/benchmark4pipeline_c

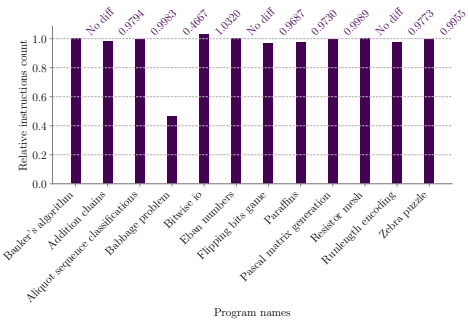


Figure 2: Vertical bars show the relative binary size in # of instructions. The smaller, the better.

4.1 Methodology

To evaluate our superoptimization pipeline, we run it on each program with four Souper configurations:

(1) Inferring only replacements for constant values
(2) Inferring replacements with no more than 2 instructions, i.e. a new replacement is composed by no more than two instructions
(3) CEGIS (Counter Example Guided Inductive Synthesis, algorithm developed by Gulwani et al. [9])
(4) Enumerative synthesis with no replacement size limit

In the rest of the paper, we report on the best configuration per program. Our appendix website contains the results for all configurations and all programs.

With respect to correctness, we rely on Souper's verification to check that every replacement on each program is correct. That means that the superoptimized programs are semantically equivalent. Every candidate search is done with a 300 seconds timeout. For each program, we report the best optimized case over all mentioned configurations. To discuss the results, we report the relative instruction count before and after superoptimization.

For the baseline program, we ask LLVM to generate WASM programs based on the 'wasm32-unknown-unknown' target with the -O3 optimization level. Our experiments run on an Azure machine with 8 cores (16 virtual CPUs) at 3.20GHz and 64GB of RAM.

4.2 Results

Figure 2 shows the relative size improvement with superoptimization. The median size reduction is 0.33% of the original instruction count over the tested programs. From the 12 tested programs, 8 have been improved using our pipeline whereas 3 have no changes and 1 is bigger (**Bitwise IO**). The most superoptimized program is **Babbage problem**, for which the resulting code after superoptimization is 46.67% smaller than the baseline version.

We now discuss the **Babbage problem** program, originally written in 15 lines of C code[3]. The pipeline found 3 successful code replacements for superoptimization out of 7 candidates. The best

[3] http://www.rosettacode.org/wiki/Babbage_problem#C

superoptimized version contains 21 instructions, which is much less than the original which has 45 instructions. The superoptimization code difference program is shown in Figure 3. Our pipeline, using Souper, finds that the loop inside the program can be replaced with a **`const`** value in the top of the stack, see lines 8 and 12 in Figure 3. The value, 25264, is the solution to the Babbage problem. In other terms, the superoptimization pipeline has successfully symbolically executed the problem.

The **Babbage problem** code is composed of a loop which stops when it discovers the smaller number that fits with the Babbage condition below.

```
while((n * n) % 1000000 != 269696) n++;
```

In theory, this value can also be inferred by unrolling the loop the correct number of times with llvm-opt. However, llvm-opt cannot unroll a **`while`**-loop because the loop count is not known at compile time. Additionally, this is a specific optimization that does not generalize well when optimizing for code size and requires a significant amount of time per loop.

On the other hand, Souper can deal with this case. The variable that fits the Babbage condition is inferred and verified in the SMT solver. Therefore the condition in the loop will always be false, resulting in dead code that can be removed in the final stage that generates WASM from bitcode.

In the case of the **Bitwise IO** program, we observe an increase in the number of instructions after superoptimization. From the original number of 875 instructions, the resulting count after the Souper pass is increased to 903 instructions. In this case, Souper finds 4 successful replacements out of 207 possible ones. Looking at the changes, it turns out that the LLVM IR code costs less than the original following the Souper cost function. However, the WebAssembly LLVM backend (wasm-ld tool) that transforms LLVM to WASM creates a longer WASM version. This a consequence of the discussion on Souper in subsection 2.3. In practice, it is straightforward to detect and discard those cases.

4.3 Correctness Checking

To validate the correctness of the superoptimized program we perform a comparison of the output of the non-superoptimized program and the superoptimized one. For 7/12 programs, both versions, non-superoptimized and superoptimized, behave equally and return the expected output. For 5/12 programs we cannot run them because the code generated for the target WASM architecture lacks required runtime primitives.

5 RELATED WORK

Our work spans the areas of compilation, transformation, optimization and web programming. Here we discuss three of the most relevant works that investigate superoptimization and web technologies.

Churchill et al. [4] use STOKE [1] to superoptimize loops in large programs such as the Google Native Client [3]. They use a bounded verifier to make sure that every generated optimization goes through all the checks for semantic equivalence. We apply the concept of superoptimization to the same context, but with a different stack, WebAssembly. Also, our work offloads the problem

```
1    (func $__original_main (type 2) (result i32)
2  -   (local i32 i32 i32 i32)
3  +   (local i32)
4      global.get 0
5      i32.const 16
6      i32.sub
7      local.tee 0
8  -   {...Removed code}
9      global.set 0
10     local.get 0
11 -   local.get 1
12 +   i32.const 25264
13     i32.store
14     i32.const 1024
15     local.get 0
16     call $printf
17     drop
18     local.get 0
19     i32.const 16
20     i32.add
21     global.set 0
22     i32.const 0)
```

```
1  -   i32.const -1
2  -   local.set 1
3  -   block  ;; label = @1
4  -     loop  ;; label = @2
5  -       local.get 1
6  -       i32.const 1
7  -       i32.add
8  -       local.tee 1
9  -       local.get 1
10 -       i32.mul
11 -       local.tee 2
12 -       i32.const 1000000
13 -       i32.rem_u
14 -       local.set 3
15 -       local.get 2
16 -       i32.const 2147483647
17 -       i32.eq
18 -       br_if 1 (;@1;)
19 -       local.get 3
20 -       i32.const 269696
21 -       i32.ne
22 -       br_if 0 (;@2;)
23 -     end
24 -   end
```

Figure 3: Output of superoptimization WASM bytecode for the Babbage problem program.

of semantic checking to an SMT solver, included in the Souper internals.

Emscripten is an open source tool for compiling C/C++ to the Web Context. Emscripten provides both, the WASM program and the JavaScript glue code. It uses LLVM to create WASM but it provides support for faster linking to the object files. Instead of all the IR being compiled by LLVM, the object file is pre-linked with WASM, which is faster. The last version of Emscripten also uses the WASM LLVM backend as the target for the input code.

To our knowledge, at the time of writing, the closest related work is the "souperify" pass of Binaryen [14]. It is implemented as an additional analysis on top of the existing ones. Compared to our pipeline, Binaryen does not synthesize WASM code from the Souper output.

6 CONCLUSION

We propose a pipeline for superoptimizing WebAssembly. It is a principled integration of two existing tools, LLVM and Souper, that provides equivalent and smaller WASM programs.

We have shown that the superoptimization pipeline works on a benchmark of 12 WASM programs. As for other binary formats, superoptimization of WebAssembly can be seen as complementary to standard optimization techniques. Our future work will focus on extending the pipeline to source languages that are not handled, such as TypeScript and WebAssembly itself.

ACKNOWLEDGEMENT

This work has been partially supported by WASP program and by the TrustFull project financed by the Swedish Foundation for Strategic Research. We thank John Regehr and the Souper team for their support.

REFERENCES

[1] Sorav Bansal and Alex Aiken. 2006. Automatic Generation of Peephole Superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS XII)*. Association for Computing Machinery, New York, NY, USA, 394–403. https://doi.org/10.1145/1168857.1168906

[2] Rudy Bunel, Alban Desmaison, M. Pawan Kumar, Philip H. S. Torr, and Pushmeet Kohli. 2016. Learning to superoptimize programs. *arXiv e-prints* 1, 1, Article arXiv:1611.01787 (Nov. 2016), 10 pages. arXiv:cs.LG/1611.01787

[3] Google Chrome. 2013. Welcome to Native Client - Google Chrome. Retrieved Dec 27, 2019 from https://developer.chrome.com/native-client

[4] Berkeley Churchill, Rahul Sharma, JF Bastien, and Alex Aiken. 2017. Sound Loop Superoptimization for Google Native Client. *SIGPLAN Not.* 52, 4 (April 2017), 313–326. https://doi.org/10.1145/3093336.3037754

[5] Emscripten Community. 2015. emscripten-core/emscripten. Retrieved 2019-12-11 from https://github.com/emscripten-core/emscripten

[6] LLVM community. 2019. LLVM 10 documentation. Retrieved 2019-12-12 from http://llvm.org/docs/

[7] World Wide Web Consortium. 2016. WebAssembly becomes a W3C Recommendation. Retrieved Dec 5, 2019 from https://www.w3.org/2019/12/pressrelease-wasm-rec.html

[8] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[9] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of Loop-Free Programs. *SIGPLAN Not.* 46, 6 (June 2011), 62–73. https://doi.org/10.1145/1993316.1993506

[10] Robbert Gurdeep Singh and Christophe Scholliers. 2019. WARDuino: A Dynamic WebAssembly Virtual Machine for Programming Microcontrollers. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (Athens, Greece) *(MPLR 2019)*. ACM, New York, NY, USA, 27–36. https://doi.org/10.1145/3357390.3361029

[11] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web up to Speed with WebAssembly. *SIGPLAN Not.* 52, 6 (June 2017), 185–200. https://doi.org/10.1145/3140587.3062363

[12] Massalin Henry. 1987. Superoptimizer: a look at the smallest program. *ACM SIGARCH Computer Architecture News* 15, 5 (Nov 1987), 122–126. https://doi.org/10.1145/36177.36194

[13] LLVM. 2019. WebAssembly lld port — lld 10 documentation. https://lld.llvm.org/WebAssembly.html

[14] WebAssembly. Development of WebAssembly and associated infrastructure. 2017. emscripten-core/emscripten. Retrieved 2019-12-11 from https://github.com/WebAssembly/binaryen

[15] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Gratian Lup, Jubi Taneja, and John Regehr. 2017. Souper: A Synthesizing Superoptimizer. *arXiv e-prints* 2, 1, Article arXiv:1711.04422 (Nov. 2017), 10 pages. arXiv:cs.PL/1711.04422

[16] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic Superoptimization. In *Proceedings ASPLOS'13*. ACM, New York, NY, USA, 305–316. event-place: Houston, Texas, USA.

CROW: CODE DIVERSIFICATION FOR WEBASSEMBLY

Javier Cabrera-Arteaga, Orestis Floros, Oscar Vera-Pérez, Benoit Baudry, Martin Monperrus
Network and Distributed System Security Symposium (NDSS 2021), MADWeb

https://doi.org/10.14722/madweb.2021.23004

CROW: Code Diversification for WebAssembly

Javier Cabrera Arteaga
KTH Royal Institute of Technology
Stockholm, Sweden
javierca@kth.se

Orestis Floros
KTH Royal Institute of Technology
Stockholm, Sweden
forestis@kth.se

Oscar Luis Vera Perez
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
oscar.vera-perez@inria.fr

Benoit Baudry
KTH Royal Institute of Technology
Stockholm, Sweden
baudry@kth.se

Martin Monperrus
KTH Royal Institute of Technology
Stockholm, Sweden
martin.monperrus@csc.kth.se

***Abstract*—The adoption of WebAssembly increases rapidly, as it provides a fast and safe model for program execution in the browser. However, WebAssembly is not exempt from vulnerabilities that can be exploited by malicious observers. Code diversification can mitigate some of these attacks. In this paper, we present the first fully automated workflow for the diversification of WebAssembly binaries. We present CROW, an open-source tool implementing this workflow through enumerative synthesis of diverse code snippets expressed in the LLVM intermediate representation. We evaluate CROW's capabilities on 303 C programs and study its use on a real-life security-sensitive program: libsodium, a modern cryptographic library. Overall, CROW is able to generate diverse variants for 239 out of $303\,(79\%)$ small programs. Furthermore, our experiments show that our approach and tool is able to successfully diversify off-the-shelf cryptographic software (libsodium).**

I. Introduction

WebAssembly is the fourth official language of the Web [36]. The language provides low-level constructs enabling efficient execution times, much closer to native code than JavaScript. It constitutes a fast and safe platform to execute programs in the browser and embedded environments [21]. Consequently, the adoption of WebAssembly has been rapidly growing since its introduction in 2015. Nowadays, languages such as Rust and C/C++ can be compiled to WebAssembly using mature toolchains and can be executed in all notable browsers.

The WebAssembly execution model is designed to be secure and to prevent many memory and control flow attacks. Still, as its official documentation admits [11], WebAssembly is not exempt from vulnerabilities that could be exploited [30]. Code diversification [5], [28] is one additional protection that can harden the WebAssembly stack. This consists in synthesizing different variants of an original program that provide the same functionalities but exhibit different execution traces. In this paper, we investigate the feasibility of diversifying WebAssembly code, which is, to the best of our knowledge, an unresearched area.

Network and Distributed Systems Security (NDSS) Symposium 2021
21-24 February 2021, San Diego, CA, USA
ISBN 1-891562-66-5
https://dx.doi.org/10.14722/madweb.2021.23xxx
www.ndss-symposium.org

Our contribution is a workflow and a tool, called CROW, for automatic diversification of WebAssembly programs. It takes as input a C/C++ program and produces a set of diverse WebAssembly binaries as output. The workflow is based on enumerative code synthesis. First, CROW lists blocks that are potentially relevant for diversification, second, CROW enumerates alternative instruction sequences, and third, CROW checks that the new instruction sequences are functionally equivalent to the original block. CROW builds on the idea of superdiversification [25] and extends the concept to the enumeration of a set of variants instead of synthesizing only one solution. We also take into account the specificities of WebAssembly and the details of its execution.

We evaluate the diversification capabilities of CROW in two ways. First, we diversify 303 small C programs compiled to WebAssembly. Second, we run CROW to diversify a real-life cryptographic library that natively supports WebAssembly. In both cases, we measure the diversity among binary code variants, as well as the diversity of execution traces. When measuring the diversity in binary code, we compare the WebAssembly and the machine code variants. This way we assess the ability of CROW at synthesizing variations in WebAssembly, as well as the extent to which these variations are preserved when compiling WebAssembly to machine code. Our original experiments demonstrate the feasibility of diversifying WebAssembly code. CROW generates diverse variants for $239/303\,(79\%)$ C programs. TurboFan, the optimizing compiler used in the V8 engine, preserves 99.48% of these variants. CROW successfully synthesizes variants for the cryptographic library. The variants indeed yield either different execution traces. This is promising milestone in getting a more secure Web environment through diversification.

To sum up, our contributions are:

- CROW: the first automated workflow and tool to diversify WebAssembly programs, it generates many diverse WebAssembly binaries from a single input program.
- A quantitative evaluation over 303 programs showing the capability of CROW to diversify WebAssembly binaries and measuring the impact of diversification on execution traces.
- A feasibility study of the diversification on a real-world WebAssembly program, demonstrating that CROW can handle libsodium, a state-of-the-art cryptographic library.

II. Background

A. WebAssembly

WebAssembly is a binary instruction format for a stack-based virtual machine. It is designed to address the problem of safe, fast, portable and compact low-level code on the Web. The language was first publicly announced in 2015 and since then, most major web browsers have implemented support for the standard. Besides the Web, WebAssembly is independent of any specific hardware or languages and can run in a standalone Virtual Machine (VM) or in other environments such as Arduino [20]. A paper by Haas et al. [21] formalizes the language and its type system, and explains the design rationale.

Listing 1 and 2 illustrate WebAssembly. Listing 1 presents the C code of two functions and Listing 2 shows the result of compiling these two functions into a WebAssembly module. The `type` directives at the top of the module declare the function: the types of its parameters and the type of the result. Then, the definitions for the function follow. These definitions are sequences of stack machine instructions. At the end, the `main` function is exported so that it can be called from outside this WebAssembly module, typically from JavaScript. WebAssembly has four primitive types: integers (`i32` and `i64`) and floats (`f32` and `f64`) and it includes structured instructions such as `block`, `loop` and `if`.

Listing 1: C function that calculates the quantity $2x + x$

```
int f(int x) { return 2 * x + x; }

int main(void) { return f(10); }
```

Listing 2: WebAssembly code for Listing 1.

```
(module
  (type (;0;) (func (param i32) (result i32)))
  (type (;1;) (func (result i32)))
  (func (;0;) (type 0) (param i32) (result i32)
    local.get 0
    local.get 0
    i32.const 2
    i32.mul
    i32.add)
  (func (;1;) (type 1) (result i32)
    i32.const 10
    call 0)
  (export "main" (func 1)))
```

WebAssembly is characterized by an extensive security model [11] founded on a sandboxed execution environment that provides protection against common security issues such as data corruption, code injection and return oriented programming (ROP). However, WebAssembly is no silver bullet and is vulnerable under certain conditions [30]. This motivates our work on software diversification as one possible mitigation among the wide range of security counter-measures.

B. Motivation for Moving Target Defense in the Web

The distribution model for web computing is as follows: build one binary and distribute millions of copies, all over the world, which run on browsers. In this model an attacker has two key advantages over the developers: she has a runtime environment that she fully controls and observes in any possible way. Consequently, when she finds a flaw in this virtually transparent environment, knowing that this flaw is present in the millions of copies that have been distributed over the world, she can exploit the flaw at scale.

The developers can never assume that they can control the web browser. Yet, they can challenge the second advantage of the attacker, known as the break-once-break-everywhere advantage. The developers can stop distributing clones of the binary and distribute diverse versions instead, as suggested by the pioneering software diversification works of Cohen [12] and Forrest et al. [19].

In the context of diversification, moving target defense [40] means distributing diverse variants constantly. In the context of the web, it means distributing a different variant at each HTTP request. Moving target defense is appropriate for mitigating yet unknown vulnerabilities. The diversification technique does not always remove the potential flaws, yet the vulnerabilities in the diversified binaries can be located in different places. With moving target defense, a successful attack on one browser cannot be performed on another browser with the same effectiveness. The diversified binaries that CROW outputs can be used interchangeably over the network, in a moving target defence choreographed over the web.

To sum up, by combining moving target defense deployment to diversification, we reduce the information asymmetry between the Web attacker and the defender, increasing the uncertainty and complexity of successful attacks over all client browsers [16], [42].

III. CROW's Diversification Technique

In this section we describe the workflow of CROW for diversifying WebAssembly programs. First we introduce the main concepts behind CROW. Then, we describe each stage of the workflow and we discuss the key implementation details.

A. Definitions

In this subsection we define the key concepts for CROW.

Definition 1: Block (based on Aho et al. [2]): Let P be a program. A block B is a grouping of declarations and statements in P inside a function F.

Definition 2: Program state (based on Mangpo et al. [35]): At any point in time, the program state S is defined as the collection of local and global variables, and, the program counter pointing to the next instruction.

Definition 3: Pure block: A block B is said to be pure if and only if, given the program state S_i, every execution of B produces the same state S_o.

Definition 4: Functional equivalence modulo program state (based on Le et al. [29]): Let B_1 and B_2 be two blocks. We consider the program state before the execution of the block, S_i, as the input and the program state after the execution of the block, S_o, as the output. B_1 and B_2 are functionally equivalent if given the same input S_i both codes produce the same output S_o.

Definition 5: Code replacement: Let P be a program and T a pair of blocks (B_1, B_2). T is a candidate code replacement if B_1 and B_2 are both pure as defined in Definition 3 and functionally equivalent as defined in Definition 4. Applying T to P means replacing B_1 by B_2. The application of T to P produces a program variant P' which consequently is functionally equivalent to P.

CROW generates new program variants by finding and applying code replacements as defined in Definition 5. A program variant could be produced by applying more than one candidate code replacement. For example, the tuple, composed by the code blocks in Listing 3 and Listing 4, is a code replacement for Listing 2.

Listing 3: WebAssembly pure code block from Listing 2.

```
local.get 0
i32.const 2
i32.mul ; 2 * x ;
```

Listing 4: Code block that is functionally equivalent to Listing 3

```
local.get 0
i32.const 1
i32.shl ; x << 1 ;
```

B. Overview

CROW synthesizes variants for WebAssembly programs. We assume that the programs are generated through the LLVM compilation pipeline. This assumption is motivated as follows: first, LLVM-based compilers are the most popular compilers to build WebAssembly programs [30]; second, the availability of source code (typically C/C++ for WebAssembly) provides a structure to perform code analysis and produce code replacements that is richer than the binary code.

CROW takes as input a C/C++ program and produces a set of unique, diversified WebAssembly binaries. Figure 1 shows the stages of this workflow. The workflow starts with compiling the input program into LLVM bitcode using clang. Then, CROW analyzes the bitcode to identify all pure blocks and to synthesize a set of candidate replacements for each pure block. This is what we call the *exploration* stage. In the *generation* stage, CROW combines the candidate code replacements to generate different LLVM bitcode variants. Finally, those bitcode variants are compiled to WebAssembly binaries that can be sent to web browsers.

Challenges. The concept of diversifying WebAssembly programs is novel and it is arguably hard for the following reasons. First, WebAssembly is a structured binary format, without goto-like instructions. This prevents the direct application of a wide range of diversification operators based on goto [41]. Second, the existing transformation and diversification tools target instruction sets larger than the one of WebAssembly [39]. This limits the efficiency of diversification, and the possibility of searching for a large number of equivalent code replacements. We address the former challenge using the LLVM intermediate representation as the target for diversification. We address the latter challenge by tailoring a superoptimizer for LLVM, using its subset of the LLVM intermediate representation. In particular, we prevent the superoptimizer from synthesizing instructions that have no correspondence in WebAssembly (for example, `freeze` instructions), which is an essential step to get executable diversified WebAssembly code.

C. Exploration stage

Given a program P for which we want to generate WebAssembly variants, the exploration stage of CROW identifies all pure blocks in the LLVM bitcode of P. CROW considers every directed acyclic graph contained in one function as a pure block. Then, CROW searches for code replacements for each one of them.

The generation of a code replacement consists of two steps. First, the synthesis of the new block, and, second, equivalence checking. Every variant block that passes the equivalence check is stored for use in diversification. The synthesis of block variants consists of enumerating all possible blocks that can be built as a combination of a given number of instructions, bounded by a maximum value to keep a tractable synthesis space.

There are two parameters to control the size of the search space and hence the time required to traverse it. On one hand, one can limit the size of the variants. In our experiments we limit the block variants to a maximum of 50 instructions. On the other hand, one can limit the set of instructions that are used for the synthesis. In our experiments, we use between 1 instruction (only additions) and 60 instructions (all supported instructions in the synthesizer). This configuration allows the user to find a trade-off between the amount of variants that are synthesized and the time taken to produce them.

Listing 5: Listing 1 in LLVM's intermediate representation.

```
define i32 @f(i32) {
 %2 = mul nsw i32 %0,2
 %3 = add nsw i32 %0,%2

 ret i32 %3
}

define i32 @main() {
 %1 = tail call i32 @f(i32 10)
 ret i32 %1
}
```

Block A

```
%2 = mul nsw i32 %0,2
```

Block B

```
%2 = mul nsw i32 %0,2
%3 = add nsw i32 %0,%2
```

In Listing 5 we illustrate the LLVM bitcode representation of Listing 1. In this bitcode, CROW identifies two pure blocks in function `f()`, which are displayed on the right part of the listing, in gray and green. The first pure block is composed of one single instruction (line 2) that performs the `2*x` multiplication. The second block has two instructions, one multiplication and one addition.

Using CROW, it is possible to diversify both blocks. For example, using a maximum of 1 instruction per replacement and searching over the complete bitcode instruction set, a potential replacement for Block A is: `%2 = shl nsw i32 %0,1 %`. This replacement calculates the same expression `2*x`, using a shift left operation.

To determine the equivalence between a pure block and a candidate replacement, we use an equivalence checker based on SMT [17]. In our example, the checker would prove that there cannot be a value of x such that $2 * x \neq x \ll 1$. In general, if no such counter-example exists, then the functional equivalence is assumed. On the other hand, if there exists an input resulting in different outputs for a block and a variant, then they are proven not equivalent and the variant is discarded.

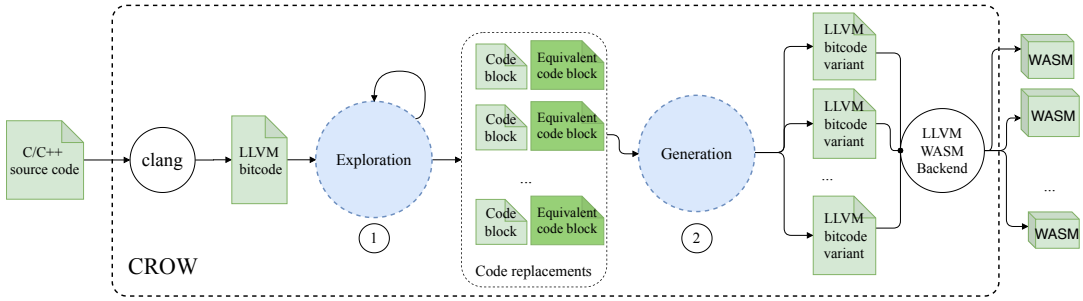


Fig. 1: CROW's workflow for diversifying WebAssembly programs.

D. Generation stage

In this stage, we select and combine code replacements that have been synthesized during the exploration stage, in order to generate WebAssembly binary variants. We apply each code replacement to the original program to produce a LLVM IR variant. Then, this IR is compiled into a WebAssembly binary. CROW generates WebAssembly binaries from all possible combinations of code replacements as the power set over all code replacements.

After the exploration phase, it is possible that two subsets of code replacements overlap, *i.e.,* they produce the same WebAssembly binary. The overlap between blocks is explained as follows: Let $S = \{(B_1, R_1), (B_1, R_2), \cdots, (B_n, R_m)\}$ be a set of candidate replacements over a program P. If two blocks from the original program B_i, B_j, $j \neq i$, overlap, *i.e.,* the intersection of $\mathrm{CFG}(B_i)$[1] and $\mathrm{CFG}(B_j)$ is not empty, then only the replacements of the largest original block are preserved when combining blocks.

In this example, the exploration stage synthesizes $6 + 1$ bitcode variants for the considered blocks respectively, which results in 14 module variants (the power set combination size). Yet, the generation stage would eventually generate 7 variants from the original WebAssembly binary. This gap between the number of potential and the actual number of variants is a consequence of the redundancy among the bitcode variants when composing several variants into one.

E. Implementation

The majority of the WebAssembly applications are built from C/C++ source code using the LLVM toolchain. Consequently, the implementation of CROW is based on LVVM/ Furthermore, CROW extends Souper [38], a superoptimizer for LLVM that aims to reduce the size of binary code. Souper has its own intermediate representation, which is a subset of the LLVM IR.

To extract code blocks, we scan LLVM modules, looking for instructions that return integer-typed values. Each such instruction is considered as the exit of a code block. Souper's representation of a code block is built as a backward traversal process through the dependencies of the detected instruction. If memory loads or function calls are found, the backward traversal process is stopped and the current instruction is considered as an input variable for the code block. Notice that, by construction, Souper's translation is oblivious to the memory model, thus, it cannot infer string data types or other abstract data types. The translation from Souper IR to a BitVector SMT theory is done on the fly. Souper uses the z3[2] solver to check the equivalence between a code block original and a potential replacement for it.

We now summarize the main changes that we implement in Souper and in the LLVM backend in order to support diversification. Souper, as a superoptimizer, aims at generating a single variant that is smaller than the original, yet we want to obtain as many blocks as possible. To achieve automatic diversification, we modify Souper to disable the key cost restriction functions, data-flow pruning and peephole optimizations, all being detrimental for diversification. In order to increase the number of variants that CROW can generate, CROW parallelizes the process of replacement synthesis.

In addition, CROW orchestrates a series of Souper executions with various configurations (in particular the size of the replaced expression). Finally, we carefully fine-tune a set of 19 Souper options to ensure that the search is effective for diversification in feasible time.

In the generation stage of CROW, we also modify Souper to amplify the generation of WebAssembly binary diversity. Initially, Souper generates a single bitcode variant, inserting all replacements at once. We modify it so that we can obtain a combination of code replacements. Finally, on the LLVM side, we disable all peephole optimizations in the WebAssembly backend, in particular instructions merging and constant folding. This aims to preserve the variations introduced in the LLVM bitcode during the generation of binaries.

The implementation of CROW is publicly available for sake of open science and can be reviewed at https://github.com/KTH/slumps/tree/master/crow.

IV. Evaluation Protocol

To evaluate the capabilities of CROW to diversify WebAssembly programs, we formulate the following research

[1] $\mathrm{CFG}(A)$ refers to backward Control Flow Graph starting at inst. A.

[2] https://github.com/Z3Prover/z3

questions:

RQ1: **To what extent are the program variants generated by CROW statically different?** We check whether the WebAssembly binary variants produced by CROW are different from the original WebAssembly binary. Then, we assess whether the generation of x86 machine code performed by V8's WebAssembly engine preserves CROW's transformations.

RQ2: **To what extent are the program variants generated by CROW dynamically different?** It is known that not all diversified programs produce distinguishable executions [15], sometimes it is impossible to observe different behaviors between variants. We check for the presence of different behaviors with a custom WebAssembly interpreter, characterizing the behavior of a WebAssembly program by its stack operation trace.

RQ3: **To what extent can CROW be applied to diversify real-world security-sensitive software?** We assess the ability of CROW to diversify a state-of-the-art cryptographic library for WebAssembly, libsodium [18].

A. Corpus

We answer RQ1 and RQ2 with a corpus of programs appropriate for our experiments. We take programs from the Rosetta Code project[3]. This website hosts a curated set of solutions for specific programming tasks in various programming languages. It contains a wide range of tasks, from simple ones, such as adding two numbers, to complex algorithms like a compiler lexer. We first collect all C programs from Rosetta Code, which represents 989 programs as of 01/26/2020. Next, we apply a number of filters. We discard 1) all programs that do not compile with `clang`, 2) all interactive programs requiring input from users *i.e.,* invoking functions like `scanf`, 3) all programs that contain more than 100 blocks, 4) all programs without termination, 5) all programs with non-deterministic operations, for example, programs working with time or random functions. This filter produces a final set of 303 programs.

The result is a corpus of 303 C programs. These programs range from 7 to 150 lines of code and solve a variety of problems, from the *Babbage* problem to *Convex Hull* calculation.

B. Protocol for RQ1

With RQ1, we assess the ability of CROW to generate WebAssembly binaries that are different from the original program. For this, we compute a distance metric between the original WebAssembly binary and each binary generated by CROW. Since WebAssembly binaries are further transformed into machine code before they execute, we also check that this additional transformation preserves the difference introduces by CROW in the WebAssembly binary. We use the Turbofan ahead-of-time compiler of V8, with all its possible optimizations, to generate a x86 binary for each WebAssembly binary. Then, we compare the x86 version of each variant against the x86 binary corresponding to the original WebAssembly binary.

We compare the WebAssembly and machine code of each program and its variant using Dynamic Time Warping (DTW) [31]. DTW computes the global alignment between two sequences. It returns a value capturing the cost of this alignment, which is actually a distance metric, called DTW. The larger the DTW distance, the more different the two sequences are. In our case, we compare the sequence of instructions of each variant with the initial program and the other variants. We obtain two DTW distance values for each program-variant pair: one at the level of WebAssembly code and the another one at the level of x86 code. Metric 1 below defines these metrics.

Metric 1: dt_static: Given two programs P_X and V_X written in X code, $dt\_static(P_X, V_X)$, computes the DTW distance between the corresponding program instructions for representation X ($X \in \{Wasm, x86\}$). A $dt\_static(P_X, V_X)$ of 0 means that the code of both the original program and the variant is the same, i.e., they are statically identical in the representation X. The higher the value of dt_static, the more different the programs are in representation X.

We run CROW on our corpus of 303 programs. We configure CROW to run with a diversification timeout of 6 hours per program. For each program, we collect the set of generated variants. For all pairs program, variant that are different, we compute both dt_static for WebAssembly and x86 representations.

The key property we consider is as follows: if $dt\_static(P_{Wasm}, P'_{Wasm}) > 0$ and $dt\_static(P_{x86}, P'_{x86}) > 0$, this means that both programs are still different when compiled to machine code, and we conclude that V8's compiler does not remove the transformations made by CROW. Notice that, this property only makes sense between variants of the same program (including the original).

C. Protocol for RQ2

For RQ2, we compare the executions of a program and its variants for a given input. In this experiment, we characterize the execution of a WebAssembly binary according to its trace of stack operations.

This method of tracing allows us to evaluate CROW's effect on program execution according to the WebAssembly specification, independently of any specific engine.

For each execution of a WebAssembly program, we collect a trace of stack operations. These traces are composed of stack-type instructions: `push <value>` and `pop <value>`. All traces are ordered with respect to the timestamp of the events. We compare the traces of the original program against those of the variants with DTW. DTW computes the global alignment between two traces and provides a value for the cost of this alignment.

Metric 2: dt_dyn: Given a program P and a CROW generated variant P', dt_dyn(P,P'), computes the DTW distance between the corresponding stack operation traces collected during their execution. A dt_dyn of 0 means that both traces are identical. The higher the value, the more different the stack operation traces.

To answer RQ2 we compute Metric 2 for a study subject program and all the unique program variants generated by CROW in a pairwise comparison. The pairwise comparison

[3] http://www.rosettacode.org/wiki/Rosetta_Code

allows us to compare the diversity between variants as well. We use SWAM[4] to collect the stack operation traces. SWAM is a WebAssembly interpreter that provides functionalities to capture the dynamic information of WebAssembly program executions including the stack operations. We compute the DTW distances with STRAC [10].

The builtin WebAssembly API for JavaScript is usually mutable, thus, the same model for traces collection can be implemented on top of V8. In other words, a custom interpreter can be implemented in order to collect the traces in the browser or standalone JavaScript engines. This validates the usage of SWAM to study the traces diversity.

D. Protocol for RQ3

In RQ3, we assess the ability of CROW to diversify a mature and complex software library related to security. We choose the libsodium [18] cryptographic library, which natively compiles to WebAssembly. With 3752 commits contributed by 96 developers, its API provides the basic blocks for encryption, decryption, signatures and password hashing. We experiment with code revision 2b5f8f2b, which contains 45232 lines of C code. Libsodium has 102 separate WebAssembly modules that we use as input for CROW. Each module corresponds to one C file that encompasses a set of related functions.

To answer RQ3, we run CROW on the libsodium bitcodes, generating a set of WebAssembly variants. Then, we assess both binary code diversity and behavioural diversity between the variants and the original libsodium, using the same techniques as in RQ1 and RQ2.

Collecting traces The libsodium repository includes an extensive test suite of 77 tests, where one test is one usage scenario. We use this test suite to measure the trace diversity among program variants. Since some test traces are larger than 1 GB each, we focus on reasonably sized tests: we select the $41/77$ test cases that produce a trace containing less than 50 million events each.

To measure the relative trace diversification for each test, we normalize the dt_dyn used in RQ2 by dividing it with the length of the original trace. This allows us to compare the relative success of CROW's diversification technique across different tests.

Since libsodium uses a pseudo-number generator, we set a static seed when executing libsodium, so that the diversity observed in traces is only due to CROW's diversification. This seed is given to the `arc4random` API used by libsodium in WebAssembly. To quantify the effectiveness of our diversification technique, we compare the trace distance produced by our technique with the trace distance that occurs when the seed is changed (baseline).

V. Experimental Results

In this section we present the results for the research questions formulated in section IV.

[4]https://github.com/satabin/swam

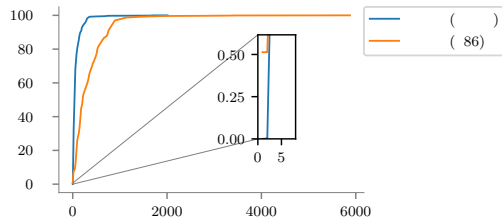


Fig. 2: Cumulative distribution for all pairwise comparisons between a program and its variants. Each line corresponds to a different program representation.

A. To what extent are the program variants generated by CROW statically different?

We run CROW on 303 C programs compiled to WebAssembly. CROW produces at least one unique program variant for $239/303$ programs. For the rest of the programs ($64/303$), the timeout is reached before CROW can find any valid variant.

We subsequently perform a manual analysis of the programs that yield more than 100 unique WebAssembly variants. This reveals one key reason that favors a large number of unique WebAssembly variants: the programs include bounded loops. In these cases CROW synthesizes variants for the loops by unrolling them. Every time a loop is unrolled, the loop body is copied and moved as part of the outer scope of the loop. This creates a new, statically different, program. The number of programs grows exponentially with nested loops.

A second key factor for the synthesis of many variants relates to the presence of arithmetic. Souper, the synthesis engine used by CROW, is effective in replacing arithmetic instructions by equivalent instructions that lead to the same result. For example, CROW generates unique variants by replacing multiplications with additions or shift left instructions (Listing 8). Also, logical comparisons are replaced, inverting the operation and the operands (Listing 9).

Listing 8: Diversification through arithmetic expression replacement.

```
local.get 0
i32.const 2
i32.mul
```

```
local.get 0
i32.const 1
i32.shl
```

Listing 9: Diversification through inversion of comparison operations.

```
local.get 0
i32.const 10
i32.gt_s
```

```
i32.const 11
local.get 0
i32.le_s
```

We now discuss the prevalence of the transformations made by CROW when the WebAssembly binaries are transformed to machine code, specifically with the V8's engine. In Figure 2 we plot the cumulative distribution of dt_static, comparing WebAssembly binaries (in blue) and x86 binaries (in orange). The figure plots a total of 103003 dt_static values for each representation, two values for each variant pair comparison (including original) for the 239 program. The value on the y-axis shows which percentage of the total comparisons lie

Listing 10: Excerpt of WebAssembly program p74: CROW replaces a loop by a constant.

```
local.set 1
loop ;; label = @1
  ...
end
...
i32.store
```

```
local.get 0
i32.const 25264
...
i32.store
```

below the corresponding dt_static value on the x-axis. Since we measure the distances between original programs and WebAssembly variants, then 100% of these binaries have $dt\_static > 0$. Let us consider the x86 variants: dt_static is strictly positive for 99.48% of variants. In all these cases, the V8 compilation phase does not undo the CROW diversification transformations. Also, we see that there is a gap between both distributions, the main reason is the natural inflation of machine code. For example, two variants that differ by one single instruction in WebAssembly, can be translated to machine code where the difference is increased by more than one machine code instruction.

The zoomed subplot focuses on the beginning of the distribution, it shows that the dt_static is zero for 0.52% of the x86 binaries. In these cases the V8 TurboFan compiler from WebAssembly to x86 reverts the CROW transformations. We find that CROW produces at least one of these reversible transformations for $34/239$ programs. Listing 11 shows one of the most common transformations that is reversed by TurboFan, according to our experiments.

Listing 11: Replacement in WebAssembly that is translated to the same x86 code by V8-TurboFan.

```
i32.const -<n>
i32.sub
```

```
i32.const <n>
i32.add
```

We look at the cases that yield a small number of variants. There is no direct correlation between the number of identified blocks and the number of unique variants. We manually analyze programs that include a significant number of pure blocks, for which CROW generates few variants. We identify two main challenges for diversification.

1) Constant computation We have observed that Souper searches for a constant replacement for more than 45% of the blocks of each program while constant values cannot be inferred. For instance, constant values cannot be inferred for memory load operations because CROW is oblivious to a memory model.

2) Combination computation The overlap between code replacements, discussed in subsection III-D, is a second factor that limits the number of unique variants. CROW can generate a high number of variants, but not all replacement combinations are necessarily unique.

Regarding the potential size overhead of the generated variants, we have compared the WebAssembly binary size of the 239 programs with their variants. The ratio of size change between the original program and the variants ranges from 82% (variants are smaller) to 125% (variants are larger) for all Rosetta programs. This limited impact on the binary size of the variants is good news because they are meant to be distributed to browsers over the network.

Answer to RQ1

CROW is able to generate diverse variants of WebAssembly programs for $239/303\,(79\%)$ programs in our corpus. We observe that programs that include bounded loops and arithmetic expressions are highly prone to diversification. V8's TurboFan compilation to x86 code preserves 99.48% of the transformations performed by CROW. To our knowledge, this is the first ever realization of automated diversification for WebAssembly.

B. To what extent are the program variants generated by CROW dynamically different?

Now, we focus on the 41 programs that have at least 9 unique WebAssembly variants in order to study the diversity of execution traces. We apply the protocol described in subsection IV-C by executing the WebAssembly programs and their unique variants in order to collect the stack operation traces. Then, we compare the traces of each pair of original program and a variant. We run 1906 program executions and we perform 98774 trace pair comparisons.

Table I summarizes the observed trace diversity, as captured by dt_dyn (Metric 2), among each program and their variants. The table is structured as follows: the first, second and third columns contain the program id, the number of unique variants and the overall sum of all blocks replacements respectively. The table summarizes the distribution of distances between stack operation trace pairs: the minimum value, the maximum value, the median value, the percentage of values equal to zero and the percentage of values greater than zero. The programs are sorted with respect to the number of unique variants. The green highlight color in $> 0\%$ columns represents more than 50% of non-zero comparisons, *i.e.,* high diversification. For instance, the first row shows the trace diversity for `p96`, where 99.70% of the pairwise comparisons between all collected traces have a different dt_dyn.

For the stack operation traces, all programs have at least one variant that produces a trace different from the original. All but one (`p81`) programs have the majority of variants producing a different stack operation trace. This shows the real effectiveness of CROW for diversifying stack operation traces.

We manually analyze variants with high and low trace diversity. We observe that constant inferring is effective at changing the stack operation trace. For instance, for program `p74` shown in Listing 10, CROW removes a loop by replacing it with a constant assignment. The execution of this variant produces traces that are different because the loop pattern is not visible anymore in the trace, and consequently, the distance between the original and the variant traces is large.

	NAME	#var	Σ	Min	Max	Median	0 %	> 0 %
1	**p96**	220	15	0	24062	820	0.30	99.70
2	p56	192	36	0	45420	1416	1.84	98.16
3	p78	159	35	0	20501	759	1.52	98.48
4	p111	144	45	0	2114	520	3.74	96.26
5	**p166**	101	152	0	44538	66	45.80	54.20
6	p122	91	34	0	46026	6434	0.24	99.76
7	p67	89	77	0	94036	85692	0.29	99.71
8	p68	85	10	0	10554	260	3.64	96.36
9	p80	78	9	0	17238	618	3.92	96.08
10	p204	77	42	0	36428	3356	0.33	99.67
11	p183	76	9	0	90628	84402	0.57	99.43
12	p136	62	70	0	62953	58028	0.60	99.40
13	p167	46	232	8	888	724	0.00	100.00
14	p226	42	13	0	90736	74476	8.26	91.74
15	p99	38	74	16	9936	5037	0.00	100.00
16	p18	36	7	0	15620	145	1.10	98.90
17	**p140**	29	17	0	13280	172	6.59	93.41
18	p59	27	6	0	85390	40	1.43	98.57
19	p199	21	87	0	27482	728	4.68	95.32
20	**p91**	21	21	0	50002	228	43.81	56.19
21	p223	21	115	16	40911	632	0.00	100.00
22	p168	20	6	0	22200	18896	2.20	97.80
23	p174	18	40	6	6566	6395	0.00	100.00
24	**p81**	17	86	0	4419	0	84.62	15.38
25	p141	17	6	8	2894	132	0.00	100.00
26	p108	16	6	0	85168	79903	8.97	91.03
27	p98	15	4	0	33	25	6.06	93.94
28	p89	14	45	10	15952	89	0.00	100.00
29	p36	14	52	312	33266	30298	0.00	100.00
30	**p135**	13	5	0	20288	20163	3.57	96.43
31	p161	12	91	240	9792	1056	0.00	100.00
32	p147	12	32	0	54071	21274	7.14	92.86
33	p11	10	38	29798	51846	35119	0.00	100.00
34	p125	10	51	0	4399	4368	7.14	92.86
35	p131	9	4	140	1454	685	0.00	100.00
36	p69	9	48	28	29243	28956	0.00	100.00
37	p134	9	20	4	514	186	0.00	100.00
38	**p74**	9	19	126	8332	6727	0.00	100.00
39	p79	9	97	4	29	16	0.00	100.00
40	p33	9	52	4	2342	15	0.00	100.00
41	p157	9	64	36	242	166	0.00	100.00

TABLE I: Dynamic diversity for 41 diversified WASM programs. The dynamic diversity is captured by dt_dyn between traces. The rows are sorted by the number of unique variants per program. The table is structured as follows: the first, second and third columns contain the program id, the number of unique variants and the overall sum of all blocks replacements respectively. Following, the stats for the dt_dyn metric. The colorized cells in the $> 0\%$ column represent high diversification.

Listing 12: Statically different WebAssembly replacements with the same behavior, gray for the original code, green for the replacement.

```
(1) i32.lt_u  i32.lt_s      (3) i32.ne        i32.lt_u
(2) i32.le_s  i32.lt_u      (4) local.get 6   local.get 4
```

We note that there is no relation between the trace distance and the number of block replacements. A high trace distance does not necessarily imply a high number of replacements. For instance, program `p135` has only 4 possible replacements overall its 5 identified blocks yet a median dt_dyn of 20163.

We subsequently analyze the cases where diversification is not reflected in stack operation traces. For example, more than 40% of the pairwise dt_dyn distances for `p166`, `p91` and `p81` are equal to zero. This indicates a lower diversity among the population of variants, than for all the other programs. This happens because some variants have two different bitcode instructions (original and replacement) that trigger the same stack operations. The instructions in Listing 12 are concrete cases of such kind of replacements. The four cases in Listing 12 leave the same value in the stack operation trace. For each case, the original instruction and the replacement are semantically equal in the program domain. The fourth case is a local variable index reallocation, this replacement only changes the index of the local variable but not the event in the stack operation trace. These replacements are sound, produce statically diverse code, but they are not useful to dynamically diversify the original program. This confirms the complementary of using static and dynamic metrics to assess diversification.

The effectiveness of CROW on diversifying stack operation traces is significant. In a security context, such diverse stack operation traces are likely to mitigate potential side-channel attacks [30]. Notably, the attacks based on code profiling are affected when the executed opcodes and the corresponding profiles are different [37].

Answer to RQ2

CROW is successful at generating diverse WebAssembly variant programs, for which we are able to observe different stack operation traces. In other words, CROW generates dynamically different binaries, and ensures that variants of a given program yield different stack operation traces.

C. To what extent can CROW be applied to diversify real-world security-sensitive software?

We run CROW on each of the 102 modules of libsodium with a 6-hour timeout. We find $45/102$ modules that do not contain any pure block, so they are not amenable to our diversification technique. CROW produces at least one valid WebAssembly module variant for 15 of the remaining 57 modules.

Module & Description	#var	#func	Diversified Functions	#calls
argon2-core Core functions for the implementation of the Argon2 key derivation (hash) function [9].	17	6	argon2_finalize argon2_free_instance argon2_initialize	0 0 0
argon2-encoding Functions for encoding and decoding (including salting) Argon2 [9] hash strings.	11	2	argon2_decode_string argon2_encode_string	0 0
blake2b-ref Reference implementation for the BLAKE2 [4] hash function.	7	11	blake2b blake2b_salt_personal blake2b_update	0 1.46E+04 2.04E+04
chacha20_ref Reference implementation of the ChaCha20 stream cipher [6].	7	5	chacha20_encrypt_bytes stream_ietf_ext_ref_xor_ic stream_ref stream_ref_xor_ic	3.51E+06 7.62E+03 1.14E+04 1.14E+05
codecs Implementations of commonly used codecs for conversions between binary formats like Base64 [26].	79	5	sodium_base642bin sodium_base64_encoded_len sodium_bin2base64 sodium_bin2hex sodium_hex2bin	0 0 0 2.57E+05 0
core_ed25519 Implementation of the Edwards-curve Digital Signature Algorithm [8].	2	19	crypto_core_ed25519_is_valid_point	0
crypto_scrypt-common Utility and low-level API functions for the scrypt key derivation (hash) function [34].	5	5	escrypt_gensalt_r	0
pbkdf2-sha256 Implementation of the Password-Based Key Derivation Function 2 (PBKDF2) [27].	14	1	escrypt_PBKDF2_SHA256	0
pwhash_scryptsalsa208sha256 High-level API for the scrypt key derivation function [34].	8	19	crypto_pwhash_scryptsalsa208sha256	0
pwhash_scryptsalsa208sha256_nosse Same as above, but does not use Streaming SIMD Extensions (SSE).	32	3	escrypt_kdf_nosse salsa20_8	0 0
randombytes Pseudorandom number generators.	1	11	randombytes_uniform	5.61E+02
salsa20_ref Contains a reference implementation of the Salsa20 stream cipher [7].	12	2	stream_ref stream_ref_xor_ic	1.14E+04 1.14E+05
scalarmult_ristretto255_ref10 Implementation of the Ristretto255 prime order elliptic curve group [22].	29	4	scalarmult_ristretto255 scalarmult_ristretto255_base scalarmult_ristretto255_scalarbytes	0 0 0
stream_chacha20 High-level API for the ChaCha20 stream cipher [8].	2	15	crypto_stream_chacha20 crypto_stream_chacha20_ietf crypto_stream_chacha20_ietf_ext crypto_stream_chacha20_ietf_ext_xor_ic crypto_stream_chacha20_ietf_xor crypto_stream_chacha20_ietf_xor_ic crypto_stream_chacha20_xor crypto_stream_chacha20_xor_ic	6.65E+02 3.19E+03 2.66E+03 1.68E+02 1.68E+02 2.32E+03 0 1.68E+02
verify Functions used to compare secrets in constant time to avoid timing attacks.	7	6	crypto_verify_16 crypto_verify_32 crypto_verify_64	2.69E+05 3.40E+03 0
Total	**256**	**114**	**40 functions**	

TABLE II: Libsodium modules with at least one variant generated by CROW. The columns on the left include the facts about each module. The first column contains the name and the functional description of the modules. The second column, *#var* (highlighted) gives the number of unique variants generated by CROW. The third column, *#func*, lists the total amount of functions in each module. The remaining columns include a list of functions that CROW has successfully diversified and the number of calls per function in the test suite.

Table II presents the key results for these 15 successfully diversified modules. The first two columns contain the name and description of the diversified module, and, the number of unique static variants. The other columns show the total number of functions inside the module, the names of the diversified functions and the number of calls to each function in the considered tests.

Generation of WebAssembly library variants from WebAssembly module variants. The successfully diversified modules can be combined to obtain a large pool of different versions of the packaged libsodium WebAssembly library. The Cartesian product of all module variants produces in theory 1.66E+15 unique libsodium variants. Yet, it is unpractical to store and execute this large number of variants. Thus, we sample the pool of possible variants to evaluate our generated variants. First, for each of the 256 modules, we rank each module variant with respect to the number of lines changed in the final WebAssembly textual format. Then, to produce the i-th library variant, we combine the i-th variant for each module of libsodium, in order to produce maximally diversified library variants first. If a module has less than i variants, we use the original, non-diversified module. According to Table II, the maximum number of unique variants for a single module is 79 (`codecs` module). Thus, we sample 79 unique libsodium variants, ordered by the amount of diversification (the first variant contains the most changes, and so on). For each variant we execute the complete test suite to validate its correctness. All test cases successfully pass for all diversified library binaries.

Dynamic evaluation of libsodium variants. We compare the dynamic behaviour of the original libsodium and the 79 library variants. Figure 3 illustrates the distribution of dt_dyn of all collected traces for each libsodium test. The dt_dyn distance is calculated between each diversified trace

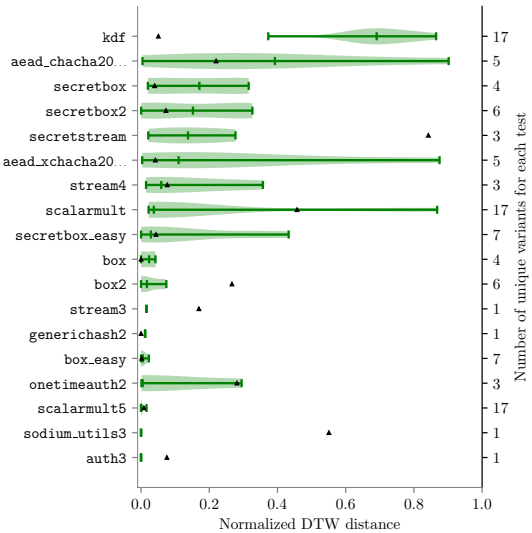


Fig. 3: Distribution of normalized dt_dyn distances over the set of libsodium variants covered by each test. The left Y axis lists the name of each test. The number of unique variants used per test is listed on the right Y axis. The black triangles point to the dt_dyn distance between two different stack operation traces of the original test with different random seeds.

and the corresponding original trace for the same test. Each horizontal bar gives the distribution of dt_dyn over the 79 diversified libraries per test. The black triangles show the dt_dyn distance between two different executions of the same test with different random seeds. They serve as a baseline to compare the artificial diversity introduced by CROW, against the natural trace diversity that appears because of random number generation.

For $18/19$ tests, we observe that CROW's diversified modules produce a different trace than the original. The wider violin plots that reach the right-hand side of the figure include variants that significantly diversify the test execution. We observe that $4/18$ tests stand out as they include variants with at least 0.8 normalized dt_dyn distance. For $6/18$ tests, there is a medium trace diversity as their dt_dyn distributions lie in the mid/left side of the plot. For the rest $8/18$ tests we observe a significantly smaller dt_dyn distance.

This means that, in the context of this cryptographic library, CROW is able to find variants that have a huge impact on the dynamic stack behaviour of the program. Meanwhile, some other replacements can have only a marginal impact during the operation of the program. One factor that can affect this is the "centrality" of the code that is being replaced. Diversified code that is called often, potentially inside loops, will have a greater impact on the stack trace of a program compared to code that is only called, for example, only during the initialization of the program.

When we compare the trace diversity against the diversity due to pseudo-number generation (black triangles in Figure 3), we observe that: for $2/18$ tests CROW trace diversification is always larger than the one due to random number generation, for $11/18$ tests there exist some variants that exhibit larger trace diversification than random number generation and for $5/18$ tests CROW trace diversification is always smaller than the one due to random number generation.

Answer to RQ3

We have successfully applied CROW to libsodium, one of the leading WebAssembly cryptography libraries. We have shown that CROW is able to create statically different variants of this real-world library, all of which being distributable to users. Our original experiments to measure the trace diversity of libsodium have proven that the generated variants exhibit significantly different execution traces compared to the original non-diversified libsodium binary. The take-away of this experiment is that CROW works on complex code.

VI. Threats to Validity

Internal: The timeout in the exploration stage is a determinant factor to generate unique variants. It is required to bound the experimental time. If the timeout is increased, the number of variants and unique variants might increase.

External: The 303 programs in our Rosetta corpus may not reflect the constructs used in the WebAssembly programs in the wild. Yet our experiment on libsodium shows that the results on the Rosetta corpus hold on real code. To increase external validity, we hope to see more benchmarks of WebAssembly programs published by the research community.

Scale: We measure behavioral diversity with DTW. We are aware that this behavioral diversity metric does not scale infinitely. To make comparisons between large execution traces, it may be necessary to use a more scalable metric. To mitigate this scale problem in future work, one option is to compare software traces using entropy analysis, as proposed by Miranskyy et al. [33].

VII. Related Work

Program diversification approaches can be applied at different stages of the development pipeline.

Static diversification: This kind of diversification consists in synthesizing, building and distributing different, functionally equivalent, binaries to end users. This aims at increasing the complexity and applicability of an attack against a large population of users [12]. Jackson et al. [24] argue that the compiler can be placed at the heart of the solution for software diversification; they propose the use of multiple semantic-preserving transformations to implement massive-scale software diversity in which each user gets their own diversified variant. Dealing with code-reuse attacks, Homescu et al. [23] propose inserting NOP instruction directly in LLVM IR to generate a variant with different code layout at each compilation. In this area, Coppens et al. [13] use compiler transformations to iteratively diversify software. The aim of their work is to prevent reverse engineering of security patches for attackers targeting vulnerable programs. Their approach, continuously applies a random

selection of predefined transformations using a binary diffing tool as feedback. A downside of their method is that attackers are, in theory, able to identify the type of transformations applied and find a way to ignore or reverse them. Our work can be extended to address this issue, providing a synthesizing solution which is more general than specific transformations.

The work closest to ours is that by Jacob et al. [25]. These authors propose the use of a "superdiversification" technique, inspired by superoptimization [32], to synthesize individualized versions of programs. In the work of Massalin, a superoptimizer aims to synthesize the shortest instruction sequence that is equivalent to the original given sequence. On the contrary, the tool developed by Jacob et al. does not output only the shortest instruction sequence, but any sequences that implement the input function. This work focuses on a specific subset of X86 instructions. Meanwhile, our approach works directly with LLVM IR, enabling it to generalize to more languages and CPU architectures. Specifically, we apply our tool on WebAssembly, something not possible with the X86-specific approach of that paper.

Runtime diversification: Previous works have attempted to generate diversified variants that are alternated during execution. It has been shown to drastically increase the number of execution traces that a side-channel attack requires to succeed. Amarilli et al. [3] are the first to propose generation of code variants against side-channel attacks. Agosta et al. [1] and Crane et al. [15] modify the LLVM toolchain to compile multiple functionally equivalent variants to randomize the control flow of software, while Couroussé et al. [14] implement an assembly-like DSL to generate equivalent code at runtime in order to increase protection against side-channel attacks. CROW focuses on static diversification of software. However, because of the specificities of code execution in the browser, this is not far from being a dynamic approach. Since WebAssembly is served at each page refreshment, every time a user asks for a WebAssembly binary, she can be served a different variant provided by CROW.

VIII. Conclusion

Security has been a major driver for the design of WebAssembly. Diversification is one additional protection mechanism that has been not yet realized for it. In this paper, we have presented CROW, the first code diversification approach for WebAssembly. We have shown that CROW is able to generate variants for a large variety of programs, including a real-world cryptographic library. Our original experiments have comprehensively assessed the generated diversity: we have shown that CROW generates diversity both among the binary code variants as well as in the execution traces collected when executing the variants. Also, we have successfully observed diverse execution traces for the considered cryptographic library, which can protect it against a range of side channel attacks.

Future work includes increasing the number of unique variants that are generated, by working on block replacement overlapping detection. Also, the exploration stage and the identification of code replacements is a highly parallelizable process, this would increase diversification performance in order to meet the demands of the internet scale.

References

[1] G. Agosta, A. Barenghi, G. Pelosi, and M. Scandale, "The MEET approach: Securing cryptographic embedded software against side channel attacks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 8, pp. 1320–1333, 2015.

[2] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. USA: Addison-Wesley Longman Publishing Co., Inc., 1986, ch. 1, pp. 28–31.

[3] A. Amarilli, S. Müller, D. Naccache, D. Page, P. Rauzy, and M. Tunstall, "Can code polymorphism limit information leakage?" in *IFIP International Workshop on Information Security Theory and Practices*. Springer, 2011, pp. 1–21.

[4] J.-P. Aumasson, S. Neves, Z. Wilcox-O'Hearn, and C. Winnerlein, "BLAKE2: simpler, smaller, fast as MD5," in *International Conference on Applied Cryptography and Network Security*. Springer, 2013, pp. 119–135.

[5] B. Baudry and M. Monperrus, "The multiple facets of software diversity: Recent developments in year 2000 and beyond," *ACM Computing Surveys (CSUR)*, vol. 48, no. 1, pp. 1–26, 2015.

[6] D. J. Bernstein, "The ChaCha family of stream ciphers," 2008. [Online]. Available: http://cr.yp.to/chacha.html

[7] ——, "The Salsa20 family of stream ciphers," in *New stream cipher designs*. Springer, 2008, pp. 84–97.

[8] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang, "High-speed high-security signatures," *Journal of cryptographic engineering*, vol. 2, no. 2, pp. 77–89, 2012.

[9] A. Biryukov, D. Dinu, and D. Khovratovich, "Argon2: new generation of memory-hard functions for password hashing and other applications," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2016, pp. 292–302.

[10] J. Cabrera Arteaga, M. Monperrus, and B. Baudry, "Scalable comparison of javascript V8 bytecode traces," in *Proceedings of the 11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*. New York, NY, USA: Association for Computing Machinery, 2019, p. 22–31.

[11] D. Chen and W3C group, "WebAssembly documentation: Security," W3C, Accessed: 18 June 2020. [Online]. Available: https://webassembly.org/docs/security/

[12] F. B. Cohen, "Operating system protection through program evolution." *Computers & Security*, vol. 12, no. 6, pp. 565–584, 1993.

[13] B. Coppens, B. De Sutter, and J. Maebe, "Feedback-driven binary code diversification," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, pp. 1–26, 2013.

[14] D. Couroussé, T. Barry, B. Robisson, P. Jaillon, O. Potin, and J.-L. Lanet, "Runtime code polymorphism as a protection against side channel attacks," in *IFIP International Conference on Information Security Theory and Practice*. Springer, 2016, pp. 136–152.

[15] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, "Thwarting cache side-channel attacks through dynamic software diversity." in *NDSS*, 2015, pp. 8–11.

[16] A. Cui and S. J. Stolfo, "Symbiotes and defensive mutualism: Moving target defense," in *Moving target defense*. Springer, 2011, pp. 99–108.

[17] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[18] F. Denis, "The Sodium cryptography library," Jun 2013. [Online]. Available: https://download.libsodium.org/doc/

[19] S. Forrest, A. Somayaji, and D. H. Ackley, "Building diverse computer systems," in *Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No. 97TB100133)*. IEEE, 1997, pp. 67–72.

[20] R. Gurdeep Singh and C. Scholliers, "WARDuino: A dynamic WebAssembly virtual machine for programming microcontrollers," in *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, ser. MPLR 2019, 2019, pp. 27–36.

[21] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with WebAssembly," in *Proceedings of the 38th*

ACM SIGPLAN Conference on Programming Language Design and Implementation, 2017, pp. 185–200.

[22] M. Hamburg, H. de Valance, I. Lovecruft, and T. Arcieri, "The ristretto group," 2017.

[23] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz, "Profile-guided automated software diversity," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2013, pp. 1–11.

[24] T. Jackson, B. Salamat, A. Homescu, K. Manivannan, G. Wagner, A. Gal, S. Brunthaler, C. Wimmer, and M. Franz, "Compiler-generated software diversity," in *Moving Target Defense*. Springer, 2011, pp. 77–98.

[25] M. Jacob, M. H. Jakubowski, P. Naldurg, C. W. N. Saw, and R. Venkatesan, "The superdiversifier: Peephole individualization for software protection," in *International Workshop on Security*. Springer, 2008, pp. 100–120.

[26] S. Josefsson, "The Base16, Base32, and Base64 data encodings," Internet Requests for Comments, RFC Editor, RFC 4648, October 2006, http://www.rfc-editor.org/rfc/rfc4648.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc4648.txt

[27] B. Kaliski, "PKCS #5: Password-based cryptography specification version 2.0," Internet Requests for Comments, RFC Editor, RFC 2898, September 2000, http://www.rfc-editor.org/rfc/rfc2898.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2898.txt

[28] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "Sok: Automated software diversity," in *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 276–291.

[29] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14, 2014, p. 216–226.

[30] D. Lehmann, J. Kinder, and M. Pradel, "Everything old is new again: Binary security of WebAssembly," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020.

[31] M. D. A. Maia, V. Sobreira, K. R. Paixão, R. A. D. Amo, and I. R. Silva, "Using a sequence alignment algorithm to identify specific and common code from execution traces," in *Proceedings of the 4th International Workshop on Program Comprehension through Dynamic Analysis (PCODA*, 2008, pp. 6–10.

[32] H. Massalin, "Superoptimizer— A Look at the Smallest Program," *ACM SIGPLAN Notices*, vol. 22, no. 10, pp. 122–126, 10 1987.

[33] A. V. Miranskyy, M. Davison, R. M. Reesor, and S. S. Murtaza, "Using entropy measures for comparison of software traces," *Information Sciences*, vol. 203, pp. 59–72, oct 2012.

[34] C. Percival, "Stronger key derivation via sequential memory-hard functions," 2009.

[35] P. M. Phothilimthana, A. Thakur, R. Bodik, and D. Dhurjati, "Scaling up superoptimization," *SIGARCH Comput. Archit. News*, vol. 44, no. 2, p. 297–310, Mar. 2016.

[36] A. Rossberg, "WebAssembly Core Specification," W3C, Tech. Rep., Dec. 2019. [Online]. Available: https://www.w3.org/TR/wasm-core-1/

[37] R. Rudd, R. Skowyra, D. Bigelow, V. Dedhia, T. Hobson, S. Crane, C. Liebchen, P. Larsen, L. Davi, M. Franz, A.-R. Sadeghi, and H. Okhravi, "Address oblivious code reuse: On the effectiveness of leakage resilient diversity," in *NDSS*, 2017.

[38] R. Sasnauskas, Y. Chen, P. Collingbourne, J. Ketema, G. Lup, J. Taneja, and J. Regehr, "Souper: A Synthesizing Superoptimizer," *arXiv preprint 1711.04422*, 2017.

[39] J. Seibert, H. Okhravi, and E. Söderström, "Information leaks without memory disclosures: Remote side channel attacks on diversified code," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 54–65.

[40] M. Taguinod, A. Doupé, Z. Zhao, and G.-J. Ahn, "Toward a moving target defense for web applications," in *2015 IEEE International Conference on Information Reuse and Integration*. IEEE, 2015, pp. 510–517.

[41] C. Wang, J. Davidson, J. Hill, and J. Knight, "Protection of software-based survivability mechanisms," in *Proc. of the Int. Conf. on Dependable Systems and Networks (DSN)*. IEEE, 2001, pp. 193–202.

[42] R. Zhuang, S. A. DeLoach, and X. Ou, "Towards a theory of moving target defense," in *Proceedings of the First ACM Workshop on Moving Target Defense*, 2014, pp. 31–40.

MULTI-VARIANT EXECUTION AT THE EDGE

Javier Cabrera-Arteaga, Pierre Laperdrix, Martin Monperrus, Benoit Baudry
Conference on Computer and Communications Security (CCS 2022), Moving Target Defense (MTD)

https://dl.acm.org/doi/abs/10.1145/3560828.3564007

Multi-variant Execution at the Edge

JAVIER CABRERA-ARTEAGA, KTH Royal Institute of technology, Sweden
PIERRE LAPERDRIX, CNRS, France
MARTIN MONPERRUS, KTH Royal Institute of Technology, Sweden
BENOIT BAUDRY, KTH Royal Institute of Technology, Sweden

Edge-Cloud computing offloads parts of the computations that traditionally occurs in the cloud to edge nodes. The binary format WebAssembly is increasingly used to distribute and deploy services on such platforms. Edge-Cloud computing providers let their clients deploy stateless services in the form of WebAssembly binaries, which are then translated to machine code, sandboxed and executed at the edge. In this context, we propose a technique that (i) automatically diversifies WebAssembly binaries that are deployed to the edge and (ii) randomizes execution paths at runtime. Thus, an attacker cannot exploit all edge nodes with the same payload. Given a service, we automatically synthesize functionally equivalent variants for the functions providing the service. All the variants are then wrapped into a single multivariant WebAssembly binary. When the service endpoint is executed, every time a function is invoked, one of its variants is randomly selected. We implement this technique in the MEWE tool and we validate it with 7 services for which MEWE generates multivariant binaries that embed hundreds of function variants. We execute the multivariant binaries on the world-wide edge platform provided by Fastly, as part as a research collaboration. We show that multivariant binaries exhibit a real diversity of execution traces across the whole edge platform distributed around the globe.

Additional Key Words and Phrases: Diversification, Moving Target Defense, Edge-Cloud computing, Multivariant execution, WebAssembly.

ACM Reference Format:
Javier Cabrera-Arteaga, Pierre Laperdrix, Martin Monperrus, and Benoit Baudry. 2022. Multi-variant Execution at the Edge. 1, 1 (August 2022), 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Edge-Cloud computing distributes a part of the data and computation to edge nodes [20, 56]. Edge nodes are servers located in many countries and regions so that Internet resources get closer to the end users, in order to reduce latency and save bandwidth. Video and music streaming services, mobile games, as well as e-commerce and news sites leverage this new type of cloud architecture to increase the quality of their services. For example, the New York Times website was able to serve more than 2 million concurrent visitors during the 2016 US presidential election with no difficulty thanks to Edge computing [4].

Authors' addresses: Javier Cabrera-Arteaga, javierca@kth.se, KTH Royal Institute of technology, Sweden; Pierre Laperdrix, pierre.laperdrix@inria.fr, CNRS, France; Martin Monperrus, martin.monperrus@kth.se, KTH Royal Institute of Technology, Sweden; Benoit Baudry, baudry@kth.se, KTH Royal Institute of Technology, Sweden.

XXXX-XXXX/2022/8-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The state of the art of edge computing platforms like Cloudflare or Fastly use the binary format WebAssembly (aka Wasm) [28, 57] to deploy and execute on edge nodes. WebAssembly is a portable bytecode format designed to be lightweight, fast and safe [17, 27]. After compiling code to a WebAssembly binary, developers spawn an edge-enabled compute service by deploying the binary on all nodes in an Edge platform. Thanks to its simple memory and computation model, WebAssembly is considered safe [44], yet is not exempt of vulnerabilities either at the execution engine's level [54] or the binary itself [38]. Implementations in both, browsers and standalone runtimes [45], have been found to be vulnerable [38, 45]. This means that if one node in an Edge network is vulnerable, all the others are vulnerable in the exact same manner. In other words, the same attacker payload would break all edge nodes at once [46]. This illustrates how Edge computing is fragile with respect to systemic vulnerabilities for the whole network, like it happened on June 8, 2021 for Fastly [3].

In this work, we introduce Multivariant Execution for WebAssembly in the Edge (MEWE), a framework that generates diversified WebAssembly binaries so that no two executions in the edge network are identical. Our solution is inspired by N-variant systems [23] where diverse variants are assembled for secretless security. Here, our goal is to drastically increase the effort for exploitation through large-scale execution path randomization. MEWE operates in two distinct steps. At compile time, MEWE generates *variants* for different functions in the program. A function variant is semantically identical to the original function but structurally different, i.e., binary instructions are in different orders or have been replaced with equivalent ones. All the function variants for one service are then embedded in a single multivariant WebAssembly binary. At runtime, every time a function is invoked, one of its variant is randomly selected. This way, the actual execution path taken to provide the service is randomized each time the service is executed, hardening Break-Once-Break-Everywhere (BOBE) attacks.

We experiment MEWE with 7 services, composed of hundreds of functions. We successfully synthesize thousands of function variants, which create orders of magnitude more possible execution paths than in the original service. To determine the runtime randomness of the embedded paths, we deploy and run the mutlivariant binaries on the Fastly edge computing platform (leading CDN platform). We collaborated with Fastly to experiment MEWE on the actual production edge computing nodes that they provide to their clients. This means that all our experiments ran in a real-world setting. For this experiment, we execute each multivariant binary several times on every edge computing node provided by Fastly. Our experiment shows that the multivariant binaries render the same service as the original, yet with highly diverse execution traces.

The novelty of our contribution is as follows. First, we are the first to perform software diversification in the context of edge computing,

with experiments performed on a real-world, large-scale, commercial edge computing platform (Fastly). Second, very few works have looked at software diversity for WebAssembly [18, 45], our paper contributes to proving the feasibility of this challenging endeavour.

To sum up, our contributions are:

- MEWE: a framework that builds multivariant WebAssembly binaries for edge computing, combining the automatic synthesis of semantically equivalent function variants, with execution path randomization.
- Original results on the large-scale diversification of WebAssembly binaries, at the function and execution path levels.
- Empirical evidence of the feasibility of deploying our novel multivariant execution scheme on a real-world edge-computing platform.
- A publicly available prototype system, shared for future research on the topic: https://github.com/Jacarte/MEWE.

This work is structured as follows. First, Section 2 present a background on WebAssembly and its usage in an edge-cloud computing scenario. Section 3 introduces the architecture and foundation of MEWE while Section 4 and Section 5 present the different experiments we conducted to show the feasibility of our approach. Section 6 details the Related Work while Section 7 concludes this paper.

2 BACKGROUND

In this section we introduce WebAssembly, as well as the deployment model that edge-cloud platforms such as Fastly provide to their clients. This forms the technical context for our work.

2.1 WebAssembly

WebAssembly is a bytecode designed to bring safe, fast, portable and compact low-level code on the Web. The language was first publicly announced in 2015 and formalized by Haas et al. [27]. Since then, most major web browsers have implemented support for the standard. Besides the Web, WebAssembly is independent of any specific hardware, which means that it can run in standalone mode. This allows for the adoption of WebAssembly outside web browsers [17], e.g., for edge computing [45].

```c
int f(int x) {
    return 2 * x + x;
}
```

Listing 1. C function that calculates the quantity $2x + x$

```
(module
  (type (;0;) (func (param i32) (result i32)))
  (func (;0;) (type 0) (param i32) (result i32)
    local.get 0
    local.get 0
    i32.const 2
    i32.mul
    i32.add)
    (export "f" (func 0)))
```

Listing 2. WebAssembly code for Listing 1.

WebAssembly binaries are usually compiled from source code like C/C++ or Rust. Listing 1 and 2 illustrate an example of a C function turned into WebAssembly. Listing 1 presents the C code of one function and Listing 2 shows the result of compiling this function into a WebAssembly module. The WebAssembly code is further interpreted or compiled ahead of time into machine code.

2.2 Web Assembly and Edge Computing

Using Wasm as an intermediate layer is better in terms of startup and memory usage, than containerization or virtualization [32, 44]. This has encouraged edge computing platforms like Cloudflare or Fastly to adopt WebAssembly to deploy client applications in a modular and sandboxed manner [28, 57]. In addition, WebAssembly is a compact representation of code, which saves bandwidth when transporting code over the network .

Client applications that are designed to be deployed on edge-cloud computing platforms are usually isolated services, having one single responsibility. This development model is known as serverless computing, or function-as-a-service [45, 53]. The developers of a client application implement the isolated services in a given programming language. The source code and the HTTP harness of the service are then compiled to WebAssembly. When client application developers deploy a WebAssembly binary, it is sent to all edge nodes in the platform. Then, the WebAssembly binary is compiled on each node to machine code. Each binary is compiled in a way that ensures that the code runs inside an isolated sandbox.

2.3 Multivariant Execution

In 2006, security researchers of University of Virginia have laid the foundations of a novel approach to security that consists in executing multiple variants of the same program. They called this “N-variant systems” [23]. This potent idea has been renamed soon after as “multivariant execution”.

There is a wide range of realizations of MVE in different contexts. Bruschi et al. [16] and Salamat et al. [50] pioneered the idea of executing the variants in parallel. Subsequent techniques focus on MVE for mitigating memory vulnerabilities [31, 41] and other specific security problems including return-oriented programming attacks [58] and code injection [52]. A key design decision of MVE is whether it is achieved in kernel space [47], in user-space [51], with exploiting hardware features [36], or even throught code polymorphism [10]. Finally, one can neatly exploit the limit case of executing only two variants [35, 43]. The body of research on MVE in a distributed setting has been less researched. Notably, Voulimeneas et al. proposed a multivariant execution system by parallelizing the execution of the variants in different machines [59] for sake of efficiency.

In this paper, we propose an original kind of MVE in the context of edge computing. We generate multiple program variants, which we execute on edge computing nodes. We use the natural redundancy of Edge-Cloud computing architectures to deploy an internet-based MVE. Next section goes into the details of our procedure to generate variants and assemble them into multivariant binaries.

3 MEWE: MULTIVARIANT EXECUTION FOR EDGE COMPUTING

In this section we present MEWE, a novel technique to synthesize multivariant binaries and deploy them on an edge computing platform.

3.1 Overview

The goal of MEWE is to synthesize multivariant WebAssembly binaries, according to the threat model presented in Section 3.2.1. The tool generates application-level multivariant binaries, without any change to the operating system or WebAssembly runtime. The core idea of MEWE is to synthesize diversified function variants providing execution-path randomization, according to the diversity model presented in Section 3.2.2.

In Figure 1, we summarize the analysis and transformation pipeline of MEWE. We pass a bitcode to be diversified, as an input to MEWE. Analysis and transformations are performed at the level of LLVM's intermediate representation (LLVM IR), as it is the best format for us to perform our modifications (see Section 3.2.3). LLVM binaries can be obtained from any language with an LLVM frontend such as C/C++, Rust or Go, and they can easily be compiled to WebAssembly. In Step (1), the binary is passed to CROW [18], which is a superdiversifier for Wasm that generates a set of variants for the functions in the binary. Step (2) packages all the variants in one single multivariant LLVM binary. In Step (3), we use a special component, called a "mixer", which augments the binary with two different components: an HTTP endpoint harness and a random generator, which are both required for executing Wasm at the edge. The harness is used to connect the program to its execution environment while the generator provides support for random execution path at runtime. The final output of Step (4) is a standalone multivariant WebAssembly binary that can be deployed on an edge-cloud computing platform. In the following sections, we describe in greater details the different stages of the workflow.

3.2 Key design choices

In this section, we introduce the main design decisions behind MEWE, starting from the threat model, to aligning the code analysis and transformation techniques.

3.2.1 Threat Model. As we describe in Section 2.2, to benefit from the performance improvements offered by edge computing, developers modularize their services into a set of WebAssembly functions. The binaries are then deployed on all the nodes provided by the edge computing platforms. However, this model of distributing the exact same WebAssembly binary on hundreds of computation nodes is a serious risk for the infrastructure: a malicious developer who manages to exploit one vulnerability in one edge location can exploit all the other locations with the same attack vector.

With MEWE, we aim to defend against an attacker that perform BOBE attacks. These attacks include but are not limited to timing specific operations [6, 11], counting register spill/reload operations to study and exploit memory [48] and performing call stack analysis. They can be performed either locally or remotely by finding a vulnerability or using shared resources in the case of a multi-tenant Edge computing server but the details of such exploitation are out of scope of this study.

3.2.2 Execution Diversification Model. MEWE is designed to randomize the execution of WebAssembly programs, via diversification transformations. Per Crane et al. those transformations are made to hinder side-channel attacks [24]. All programs are diversified with behavior preservation guarantees [18]. The core diversification strategies are: (1) *Constant Inferring*. MEWE identifies variables whose value can computed at compile time and are used to control branches. This has an effect on program execution times [15]. (2) *Call Stack Randomization*. MEWE introduces equivalent synthetic functions that are called randomly. This results in randomized call stacks, which complicates attacks based on call stack analysis [40]. (3) *Inline Expansion*. MEWE inlines methods when appropriate. This also results in different call stacks, to hinder the same kind of attacks as for call stack randomization [40]. (4) *Spills/Reloads*. By performing semantically equivalent transformations for arithmetic expressions, the number of register spill/reload operations changes. Therefore, this changes the memory accesses in the machine code that is executed, affecting the measurement of memory side-channels [48].

3.2.3 Diversification at the LLVM level. MEWE diversifies programs at the LLVM level. Other solutions would have been to diversify at the source code level [8], or at the native binary level, eg x86 [22]. However, the former would limit the applicability of our work. The latter is not compatible with edge computing: the top edge computing execution platforms, e.g. Cloudflare and Fastly, mostly take WebAssembly binaries as input.

LLVM, on the contrary, does not suffer from those limitations: 1) it supports different languages, with a rich ecosystem of frontends 2) it can reliably be retargeted to WebAssembly, thanks to the corresponding mature component in the LLVM toolchain.

3.3 Variant generation

MEWE relies on the superdiversifier CROW [18] to automatically diversify each function in the input LLVM binary (Step (1)). CROW receives an LLVM module, analyzes the binary at the function block level and generates semantically equivalent variants for each function, if they exist. A function variant for MEWE is semantically equivalent to the original (i.e., same input/output behavior), but exhibits a different internal behavior through tracing. Since the variants created by CROW are artificially synthesized from the original binary, after Step (1), they are necessarily equivalent to the original program.

3.4 Combining variants into multivariant functions

Step (2) of MEWE consists in combining the variants generated for the original functions, into into a single binary. The goal is to support execution-path randomization at runtime. The core idea is to introduce one dispatcher function per original function for which we generate variants. A dispatcher function is a synthetic function that is in charge of choosing a variant at random, every time the original function is invoked during the execution. The random invocation of different variants at runtime is a known randomization technique, for example used by Lettner et al. with sanitizers [39].

With the introduction of dispatcher function, MEWE turns the original call graph into a multivariant call graph, defined as follows.

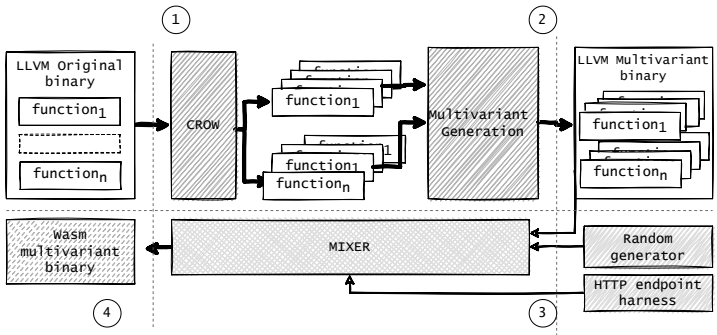


Fig. 1. Overview of MEWE. It takes as input the LLVM binary representation of a service composed of multiple functions. It first generates a set of functionally equivalent variants for each function in the binary and then generates a LLVM multivariant binary composed of all the function variants as well as dispatcher functions in charge of selecting a variant when a function is invoked. The MEWE mixer composes the LLVM multivariant binary with a random number generation library and an edge specific HTTP harness, in order to produce a WebAssembly multivariant binary accessible through an HTTP endpoint and ready to be deployed to the edge.

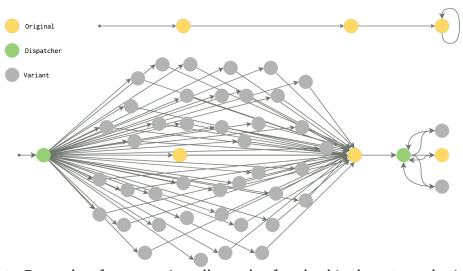


Fig. 2. Example of two static call graphs for the bin2base64 endpoint of libsodium. At the top, the original call graph, at the bottom, the multivariant call graph, which includes nodes that represent function variants (in grey), dispatchers (in green), and original functions (in yellow).

Definition 1. *Multivariant Call Graph (MCG): A multivariant call graph is a call graph* $\langle N, E \rangle$ *where the nodes in* N *represent all the functions in the binary and an edge* $(f_1, f_2) \in E$ *represents a possible invocation of* f_2 *by* f_1 *[49], where the nodes are typed. The nodes in* N *have three possible types: a function present in the original program, a generated function variant, or a dispatcher function.*

In Figure 2, we show the original static call graph for program bin2base64 (top of the figure), as well as the multivariant call graph generated with MEWE (bottom of the figure). The grey nodes represent function variants, the green nodes function dispatchers and the yellow nodes are the original functions. The possible calls are represented by the directed edges. The original bin2base64 includes 3 functions. MEWE generates 43 variants for the first function, none for the second and three for the third function. MEWE introduces two dispatcher nodes, for the first and third functions. Each dispatcher is connected to the corresponding function variants, in order to invoke one variant randomly at runtime.

The right most green node of Figure 2 is a function constructed as follows (See code in Appendix A). The function body first calls the random generator, which returns a value that is then used to invoke a specific function variant. It should be noted that the dispatcher function is constructed using the same signature as the original function.

We implement the dispatchers with a switch-case structure to avoid indirect calls that can be susceptible to speculative execution based attacks [45]. The choice of a switch-case also avoids having multiple function definitions with the same signature, which could increase the attack surface in case the function signature is vulnerable [33]. This also allows MEWE to inline function variants inside the dispatcher, instead of defining them again. Here we trade security over performance, since dispatcher functions that perform indirect calls, instead of a switch-case, could improve the performance of the dispatchers as indirect calls have constant time.

3.5 MEWE's Mixer

The MEWE mixer has four specific objectives: wrap functions as HTTP endpoints, link the LLVM multivariant binary, inject a random generator and merge all these components into a multivariant WebAssembly binary.

We use the Rustc compiler[1] to orchestrate the mixing. For the generator, we rely on WASI's specification [5] for the random behavior of the dispatchers. Its exact implementation is dependent on the platform on which the binary is deployed. For the HTTP harnesses, since our edge computing use case is based on the Fastly infrastructure, we rely on the Fastly API[2] to transform our Wasm binaries into HTTP endpoints. The harness enables a function to be called as an HTTP request and to return a HTTP response. Throughout this paper, we refer to an endpoint as the closure of invoked functions when the entry point of the WebAssembly binary is executed.

3.6 Implementation

The multivariant combination (Step ②) is implemented in 942 lines of C++ code. Its uses the LLVM 12.0.0 libraries to extend the LLVM standard linker tool capability with the multivariant generation.

[1] https://doc.rust-lang.org/rustc/what-is-rustc.html
[2] https://docs.rs/crate/fastly/0.7.3

MEWE's Mixer (Step ③) is implemented as an orchestration of the rustc and the WebAssembly backend provided by CROW. An instantiation of how the multivariant binary works can be appreciated at Appendix B. For sake of open science and for fostering research on this important topic, the code of MEWE is made publicly available on GitHub: https://github.com/Jacarte/MEWE.

4 EXPERIMENTAL METHODOLOGY

In this section we introduce our methodology to evaluate MEWE. First, we present our research questions and the services with which we experiment the generation and the execution of multivariant binaries. Then, we detail the methodology for each research question.

4.1 Research questions

To evaluate the capabilities of MEWE, we formulate the following research questions:

- **RQ1**: **(Multivariant Generation) How much diversity can MEWE synthesize and embed in a multivariant binary ?** MEWE packages function variants in multivariant binaries. With this first question, we aim at measuring the amount of diversity that MEWE can synthesize in the call graph of a program.
- **RQ2**: **(Intra MVE) To what extent does MEWE achieve multi-variant executions on an edge compute node?** With this question we assess the ability of MEWE to produce binaries that actually exhibit random execution paths when executed on one edge node.
- **RQ3**: **(Internet MVE) To what extent does MEWE achieve multivariant execution over the worldwide Fastly infrastructure?** We check the diversity of execution traces gathered from the execution of a multivariant binary. The traces are collected from all edge nodes in order to assess MVE at a worldwide scale.
- **RQ4**: **What is the impact of the proposed multi-version execution on timing side-channels?** MEWE generates binaries that embed a multivariant behavior. We measure to what extent MEWE generates different execution times on the edge. Then, we discuss how multivariant binaries contribute to less predictable timing side-channels.

The core of the validation methodology for our tool MEWE, consists in building multivariant binaries for several, relevant endpoints and to deploy and execute them on the Fastly edge-cloud platform.

4.2 Study subjects

We select two mature and typical edge-cloud computing projects to study the feasibility of MEWE. The projects are selected based on: suitability for diversity synthesis with CROW (the projects should have the ability to collect their modules in LLVM intermediate representation), suitability for deployment on the Fastly infrastructure (the project should be easily portable Wasm/WASI and compatible with the Rust Fastly API), low chances to hit execution paths with no dispatchers and possibility to collect their execution runtime information (the endpoints should execute in a reasonable time of maximum 1 second even with the overhead of instrumentation). The selected projects are: **libsodium**, an encryption, decryption, signature and password hashing library which can be ported to WebAssembly and **qrcode-rust**, a QrCode and MicroQrCode generator written in Rust.

Name	#Endpoints	#Functions	#Instr.
libsodium https://github.com/jedisct1/libsodium	5	62	6187
qrcode-rust https://github.com/kennytm/qrcode-rust	2	1840	127700

Table 1. Selected projects to evaluate MEWE: project name; the number of endpoints in the project that we consider for our experiments, the total number of functions to implement the endpoints, and the total number of WebAssembly instructions in the original binaries.

In Table 1, we summarize some key metrics that capture the relevance of the selected projects. The table shows the project name with its repository address, the number of selected endpoints for which we build multivariant binaries, the total number of functions included in the endpoints and the total number of Wasm instructions in the original binary. Notice that, the metadata is extracted from the Wasm binaries before they are sent to the edge-cloud computing platform, thus, the number of functions might be not the same in the static analysis of the project source code

4.3 Experiment's platform

We run all our experiments on the Fastly edge computing platform. We deploy and execute the original and the multivariant endpoints on 64 edge nodes located around the world[3]. These edge nodes usually have an arbitrary and heterogeneous composition in terms of architecture and CPU model. The deployment procedure is the same as the one described in Section 2.2. The developers implement and compile their services to WebAssembly. In the case of Fastly, the WebAssembly binaries need to be implemented with the Fastly platform API specification so they can properly deal with HTTP requests. When the compiled binary is transmitted to Fastly, it is translated to x86 machine code with Lucet, which ensures the isolation of the service.

4.4 RQ1 Multivariant diversity

We run MEWE on each endpoint function of our 7 endpoints. In this experiment, we bound the search for function variant with timeout of 5 minutes per function. This produces one multivariant binary for each endpoint. To answer RQ1, we measure the number of function variants embedded in each multivariant binary, as well as the number of execution paths that are added in the mutivariant call graphs, thanks to the function variants.

4.5 RQ2 Intra MTD

We deploy the multivariant binaries of each of the 7 endpoints presented in Table 2, on the 64 edge nodes of Fastly. We execute each endpoint, multiple times on each node, to measure the diversity of execution traces that are exhibited by the multivariant binaries. We have a time budget of 48 hours for this experiment. Within this

[3]The number of nodes provided in the whole platform is 72, we decided to keep only the 64 nodes that remained stable during our experimentation.

timeframe, we can query each endpoint 100 times on each node. Each query on the same endpoint is performed with the same input value. This is to guarantee that, if we observe different traces for different executions, it is due to the presence of multiple function variants. The input values are available as part of our reproduction package.

For each query, we collect the execution trace , i.e., the sequence of function names that have been executed when triggering the query. To observe these traces, we instrument the multivariant binaries to record each function entrance.

To answer RQ2, we measure the number of unique execution traces exhibited by each multivariant binary, on each separate edge node. To compare the traces, we hash them with the `sha256` function. We then calculate the number of unique hashes among the 100 traces collected for an endpoint on one edge node. We formulate the following definitions to construct the metric for RQ3.

Metric 1. *Unique traces: $R(n,e)$. Let $S(n,e) = \{T_1, T_2, ..., T_{100}\}$ be the collection of 100 traces collected for one endpoint e on an edge node n, $H(n,e)$ the collection of hashes of each trace and $U(n,e)$ the set of unique trace hashes in $H(n,e)$. The uniqueness ratio of traces collected for edge node n and endpoint e is defined as*

$$R(n,e) = \frac{|U(n,e)|}{|H(n,e)|}$$

The inputs that we pass to execute the endpoints at the edge and the received output for all executions are available in the reproduction repository at https://github.com/Jacarte/MEWE.

4.6 RQ3 Inter MTD

We answer RQ3 by calculating the normalized Shannon entropy for all collected execution traces for each endpoint. We define the following metric.

Metric 2. *Normalized Shannon entropy: $E(e)$ Let e be an endpoint, $C(e) = \cdot_{n=0}^{64} H(n,e)$ be the union of all trace hashes for all edge nodes. The normalized Shannon Entropy for the endpoint e over the collected traces is defined as:*

$$E(e) = -\Sigma \frac{p_x * log(p_x)}{log(|C(e)|)}$$

Where p_x is the discrete probability of the occurrence of the hash x over $C(e)$.

Notice that we normalize the standard definition of the Shannon Entropy by using the perfect case where all trace hashes are different. This normalization allows us to compare the calculated entropy between endpoints. The value of the metric can go from 0 to 1. The worst entropy, value 0, means that the endpoint always perform the same path independently of the edge node and the number of times the trace is collected for the same node. On the contrary, 1 for the best entropy, when each edge node executes a different path every time the endpoint is requested.

The Shannon Entropy gives the uncertainty in the outcome of a sampling process. If a specific trace has a high frequency of appearing in part of the sampling, then it is certain that this trace will appear in the other part of the sampling.

We calculate the metric for the 7 endpoints, for 100 traces collected from 64 edge nodes, for a total of 6400 collected traces per endpoint. Each trace is collected in a round robin strategy, i.e., the traces are collected from the 64 edge nodes sequentially. For example, we collect the first trace from all nodes before continuing to the collection of the second trace. This process is followed until 100 traces are collected from all edge nodes.

4.7 RQ4 Timing side-channels

For each endpoint listed in Table 2, we measure the impact of MEWE on timing. For this, we use the following metric:

Metric 3. *Execution time: For a deployed binary on the edge, the execution time is the time spent on the edge to execute the binary.*

Note that edge-computing platforms are, by definition, reached from the Internet. Consequently, there may be latency in the timing measurement due to round-trip HTTP requests. This can bias the distribution of measured execution times for the multivariant binary. To avoid this bias, we instrument the code to only measure the execution on the edge nodes.

We collect 100k execution times for each binary, both the original and multivariant binaries. We perform a Mann-Withney U test [42] to compare both execution time distributions. If the P-value is lower than 0.05, two compared distributions are different.

5 EXPERIMENTAL RESULTS

5.1 RQ1 Results: Multivariant generation

We use MEWE to generate a multivariant binary for each of the 7 endpoints included in our 2 study subjects. We then calculate the number of diversified functions, in each endpoint, as well as how they combine to increase the number of possible execution paths in the static call graph for the original and the multivariant binaries.

The sections 'Original binary' and 'Multivariant WebAssembly binary' of Table 2 summarize the key data for RQ1. In the 'Original binary' section, the first column (#F) gives the number of functions in the original binary and the second column (#Paths) gives the number of possible execution paths in the original static call graph. The 'Multivariant WebAssembly binary' section first shows the number of each type of nodes in the multivariant call graph: #Non div. is the number of original functions that could not be diversified by MEWE, #D is the number of dispatcher nodes generated by MEWE for each function that was successfully diversified, and #V is the total number of function variants generated by MEWE. The last column of this section is the number of possible execution paths in the static multivariant call graph.

For all 7 endpoints, MEWE was able to diversify several functions and to combine them in order to increase the number of possible execution paths in several orders of magnitude. For example, in the case of the `encrypt` function of libsodium, the original binary contains 23 functions that can be combined in 4 different paths. MEWE generated a total of 56 variants for 5 of the 23 functions. These variants, combined with the 18 original functions in the multivariant call graph, form 325 execution paths. In other words, the number of possible ways to achieve the same encryption function has increased from 4 to 325, including dispatcher nodes that are in charge of randomizing the choice of variants at 5 different locations of the call graph. This increased number of possible paths, combined with random choices, made at runtime, increases the effort a potential attacker needs to guess what variant is executed and hence what vulnerability she can exploit.

	Original binary		Multivariant WebAssembly binary			
Endpoint	#F	#Paths	#Non D	#D	#V	#Paths
libsodium						
encrypt	23	4	18	5	56	325
decrypt	20	3	16	5	49	84
random	8	2	6	2	238	12864
invert	8	2	6	2	125	2784
bin2base64	3	2	1	2	47	172
qrcode-rust						
qr_str	982	$688*10^6$	965	17	2092	$97*10^{12}$
qr_image	858	$1.4*10^6$	843	15	2063	$3*10^9$

Table 2. Static diversity generated by MEWE, measured on the static call graphs of the WebAssembly binaries, and the preservation of this diversity after translation to machine code. The table is structured as follows: Endpoint name; number of functions and numbers of possible paths in the original WebAssembly binary call graph; number of non diversified functions, number of created dispatchers (one per diversified functions), total number of function variants and number of execution paths in the multivariant WebAssembly binary call graph.

We have observed that there is no linear correlation between the number of diversified functions, the number of generated variants and the number of execution paths. We have manually analyzed the endpoint with the largest number of possible execution paths in the multivariant Wasm binary: `qr_str` of qrcode-rust. MEWE generated 2092 function variants for this endpoint. Moreover, MEWE inserted 17 dispatchers in the call graph of the endpoint. For each dispatcher, MEWE includes between 428 and 3 variants. If the original execution path contains function for which MEWE is able to generate variants, then, there is a combinatorial explosion in the number of execution paths for the generated Wasm multivariant module. The increase of the possible execution paths theoretically augments the uncertainty on which one to perform, in the latter case, approx. 140 000 times. As Cabrera and colleagues observed [18] for CROW, a large presence of loops and arithmetic operations in the original function code leverages to more diversification.

Looking at the #D (#Dispatchers) and #V (#Variants) columns of the 'Multivariant WebAssembly binary' section of Table 2, we notice that the number of variants generated per function greatly varies. For example, for both the `invert` and the `bin2base64` functions of Libsodium, MEWE manages to diversify 2 functions (reflected by the presence of 2 dispatcher nodes in the multivariant call graph). Yet, MEWE generates a total of 125 variants for the 2 functions in `invert`, and only 47 variants for the 2 functions in `bin2base64`. The main reason for this is related to the complexity of the diversified functions, which impacts the opportunities for the synthesis of code variations.

Columns #Non D of the 'Multivariant WebAssembly binary' section of Table 2 indicates that, in each endpoint, there exists a number of functions for which MEWE did not manage to generate variants. We identify three reasons for this, related to the diversification procedure of CROW, used by MEWE to diversify individual functions. First, some functions cannot be diversified by CROW, e.g., functions that wrap only memory operations, which are oblivious to CROW diversification technique. Second, the complexity of the function directly affects the number of variants that CROW can generate. Third, the diversification procedure of CROW is essentially a search procedure, which results are directly impacted by the tie budget for the search. In all experiments we give CROW 5 minutes maximum to synthesize function variants, which is a low budget for many functions. It is important to notice that, the successful diversification of some functions in each endpoint, and their combination within the call graph of the endpoint, dramatically increases the number of possible paths that can triggered for multivariant executions.

Answer to RQ1: MEWE dramatically increases the number of possible execution paths in the multivariant WebAssembly binary of each endpoint. The large number of possible execution paths, combined with multiple points of random choice in the multivariant call graph thwart the prediction of which path will be taken at runtime.

5.2 RQ2 Results: Intra MTD

To answer RQ2, we execute the multivariant binaries of each endpoint, on the Fastly edge-cloud infrastructure. We execute each endpoint 100 times on each of the 64 Fastly edge nodes. All the executions of a given endpoint are performed with the same input. This allows us to determine if the execution traces are different due to the injected dispatchers and their random behavior. After each execution of an endpoint, we collect the sequence of invoked functions, i.e., the execution trace. Our intuition is that the random dispatchers combined with the function variants embedded in a multivariant binary are very likely to trigger different traces for the same execution, i.e., when an endpoint is executed several times in a row with the same input and on the same edge node. The way both the function variants and the dispatchers contribute to exhibiting different execution traces is illustrated in Figure 6.

Figure 3 shows the ratio of unique traces exhibited by each endpoint, on each of the 64 separate edge nodes. The X corresponds to the edge nodes. The Y axis gives the name of the endpoint. In the plot, for a given (x,y) pair, there is blue point in the Z axis representing Metric 1 over 100 execution traces.

For all edge nodes, the ratio of unique traces is above 0.38. In 6 out of 7 cases, we have observed that the ratio is remarkably high, above 0.9. These results show that MEWE generates multivariant binaries that can randomize execution paths at runtime, in the context of an edge node. The randomization dispatchers, associated to a significant number of function variants greatly reduce the certainty about which computation is performed when running a specific input with a given input value.

Let's illustrate the phenomenon with the endpoint `invert`. The endpoint `invert` receives a vector of integers and returns its inversion. Passing a vector of integers with 100 elements as input, $I = [100, ..., 0]$, results in output $O = [0, ..., 100]$. When the endpoint executes 100 times with the same input on the original binary, we observe 100 times the same execution trace. When the endpoint is executed 100 times with the same input I on the multivariant binary, we observe between 95 and 100 unique execution traces, depending on the edge node. Analyzing the traces we observe that they include only two invocations to a dispatcher, one at the start of the trace and one at the end. The remaining events in the trace are fixed each time the endpoint is executed with the same input I. Thus, the maximum number of possible unique traces is the multiplication of the number of variants for each dispatcher, in this case $29 \times 96 = 2784$. The probability of observing the same trace is $1/2784$.

For multivariant binaries that embed only a few variants, like in the case of the `bin2base64` endpoint, the ratio of unique traces per node is lower than for the other endpoints. With the input we pass to

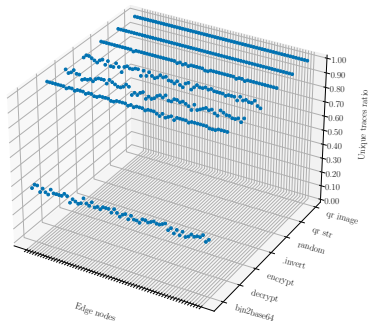


Fig. 3. Ratio of unique execution traces for each endpoint on each edge node. The X axis illustrates the edge nodes. The Y axis annotates the name of the endpoint. In the plot, for a given (x,y) pair, there is blue point representing the Metric 1 value in a set of 100 collected execution traces.

Endpoint	Entropy	MTL	σ	#ED	#RCh
libsodium					
encrypt	0.87	816	0	5	4M
decrypt	0.96	440	0	5	2M
random	0.98	15	5	2	12800
invert	0.87	7343	0	2	12800
bin2base64	0.42	57	0	1	6400
qrcode-rust					
qr_str	1.00	3045193	0	17	1348M
qr_image	1.00	3015450	0	15	1345M

Table 3. Execution trace diversity over the Fastly edge-cloud computing platform. The table is formed of 6 columns: the name of the endpoint, the normalized Shannon Entropy value (Metric 2), the median size of the execution traces (MTL), the standard deviation for the trace lengths the number of executed dispatchers (#ED) and the number of total random choices taken during all the 6400 executions (#RCh).

`bin2base64`, the execution trace includes 57 function calls. We have observed that, only one of these calls invokes a dispatcher, which can select among 41 variants. Thus, probability of having the same execution trace twice is 1/41.

Meanwhile, `qr_str` embeds thousands of variants, and the input we pass triggers the invocation of 3M functions, for which 210666 random choices are taken relying on 17 dispatchers. Consequently, the probability of observing the same trace twice is infinitesimal. Indeed, all the executions of `qr_str` are unique, on each separate edge node. This is shown in Figure 3, where the ratio of unique traces is 1 on all edge nodes.

Answer to RQ2: Repeated executions of a multivariant binary with the same input on an individual edge node exhibits diverse execution traces. MEWE successfully synthesizes multivariant binaries that trigger diverse execution paths at runtime, on individual edge nodes.

5.3 RQ3 Results: Internet MTD

To answer RQ3, we build the union of all the execution traces collected on all edge nodes for a given endpoint. Then, we compute the normalized Shannon Entropy over this set for each endpoint (Metric 2). Our goal is to determine whether the diversity of execution traces we observed on individual nodes in RQ3, actually generalizes to the whole edge-cloud infrastructure. Depending on many factors, such as the random number generator or a bug in the dispatcher, it could happen that we observe different traces on individual nodes, but that the set of traces is the same on all nodes. With RQ4 we assess the ability of MEWE to exhibit multivariant execution at a global scale.

Table 3 provides the data to answer RQ3. The second column gives the normalized Shannon Entropy value (Metric 2). Columns 3 and 4 give the median and the standard deviation for the length of the execution traces. Columns 5 and 6 give the number of dispatchers that are invoked during the execution of the endpoint (#ED) and the total number of invocations of these endpoints (#Rch). These last two columns indicate to what extent the execution paths are actually randomized at runtime. In the cases of `invert` and `random`, both have the same number of taken random choices. However, the number of variants to chose in `random` are larger, thus, the entropy, is larger than `invert`.

Overall, the normalized Shannon Entropy is above 42%. This is evidence that the multivariant binaries generated by MEWE can indeed exhibit a high degree of execution trace diversity, while keeping the same functionality. The number of randomization points along the execution paths (#Rch) is at the core of these high entropy values. For example, every execution of the `encrypt` endpoint triggers 4M random choices among the different function variants embedded in the multivariant binaries. Such a high degree of randomization is essential to generate very diverse execution traces.

The `bin2base64` endpoint has the lowest level of diversity. As discussed in RQ2, this endpoint is the one that has the least variants and its execution path can be randomized only at one point. The low level of unique traces observed on individual nodes is reflected at the system wide scale with a globally low entropy.

For both `qr_str` and `qr_image` the entropy value is 1.0. This means that all the traces that we observe for all the executions of these endpoints are different from each other. In other words, someone who runs these services over and over with the same input cannot know exactly what code will be executed in the next execution. These very high entropy values are made possible by the millions of random choices that are made along the execution paths of these endpoints.

While there is a high degree of diversity among the traces exhibited by each endpoint, they all have the same length, except in the case of `random`. This means that the entropy is a direct consequence of the invocations of the dispatchers. In the case of `random`, it naturally has a non-deterministic behavior. Meanwhile, we observe several calls to dispatchers in during the execution of the multivariant binary, which indicates that MEWE can amplify the natural diversity of traces exhibited by `random`. For each endpoint, we managed to trigger all dispatchers during its execution. There is a correlation between the entropy and the number of random choices (Column #RChs) taken during the execution of the endpoints. For a high number of dispatchers, and therefore random choices, the entropy is large, like the cases of `qr_str` and `qr_image` show. The contrary happens to `bin2base64` where its multivariant binary contains only one dispatcher.

	Original bin.		Multivariant Wasm	
Endpoint	Median (μs)	σ	Median (μs)	σ
libsodium				
encrypt	7	5	217	43
decrypt	13	6	225	47
random	16	7	232	53
invert	119	34	341	65
bin2base64	10	5	215	35
qrcode-rust				
qr_str	3,117	418	492,606	36,864
qr_image	3,091	412	512,669	41,718

Table 4. Execution time distributions for 100k executions, for the original endpoints and their corresponding multivariants. The table is structured in two sections. The first section shows the endpoint name, the median execution time and its standard deviation for the original endpoint. The second section shows the median execution time and its standard deviation for the multivariant WebAssembly binary.

Answer to RQ3: At the internet scale of the Edge platform, the multivariant binaries synthesized by MEWE exhibit a massive diversity of execution traces, while still providing the original service. It is virtually impossible for an attacker to predict which is taken for a given query.

5.4 RQ4 Results: Timing side-channels

For each endpoint used in RQ1, we compare the execution time distributions for the original binary and the multivariant binary. All distributions are measured on 100k executions. In Table 4, we show the execution time for the original endpoints and their corresponding multivariant. The table is structured in two sections. The first section shows the endpoint name, the median and standard deviation of the original endpoint. The second section shows the median and the standard deviation for the execution time of the corresponding multivariant binary.

We also observe that the distributions for multivariant binaries have a higher standard deviation of execution time. A statistical comparison between the execution time distributions confirms the significance of this difference (P-value = 0.05 with a Mann-Withney U test). This hints at the fact that the execution time for multivariant binaries is more unpredictable than the time to execute the original binary.

In Figure 4, each subplot represents the distribution for a single endpoint, with the colors blue and green representing the original and multivariant binary respectively. These plots reveal that the execution times are indeed spread over a larger range of values compared to the original binary. This is evidence that execution time is less predictable for multivariant binaries than for the original ones.

We evaluate to what extent a specific variant can be detected by observing the execution time distribution. This evaluation is based on the measurement with one endpoint. For this, we choose endpoint `bin2base64` because it is the end point that has the least variants and the least dispatchers, which is the most conservative assumption.

We dissect the collected execution times for the `bin2base64` endpoint, grouping them by execution path. In Figure 5, each opaque curve represents a cumulative execution time distribution of a unique execution path out of the 41 observed. We observe that no specific distribution is remarkably different from another one. Thus, no specific variant can be inferred out of the projection of all execution times like the ones presented in Figure 4. Nevertheless, we calculate a Mann-Whitney test for each pair of distributions, 41×41 pairs. For all cases, there is no statistical evidence that the distributions are different, $P > 0.05$.

Recall that the choice of function variant is randomized at each function invocation, and the variants have different execution times as a consequence of the code transformations, i.e., some variants execute more instructions than others. Consequently, attacks relying on measuring precise execution times of a function are made a lot harder to conduct as the distribution for the multivariant binary is different and even more spread than the original one.

We evaluate the impact of multivariant binaries on execution time. As a baseline, we consider the evaluation proposed by Fastly [1, 2]: a Markdown to HTML conversion service shall run on their edge platform and return a response in less than 100 ms, allowing one request for every single keystroke. In this context, all the multivariant binaries for Libsodium match the baseline and still support requests at the speed of keystrokes. The multivariant binaries for QR encoding respond in a reasonable time for end users, i.e., in less than half a second, but are below the baseline. In general, we note that the execution times are slower for multivariant binaries. Being under 500 ms in general, this does not represent a threat to the applicability of multivariant execution at the edge. Yet, it calls for future optimization research.

Answer to RQ4: The execution time distributions are significantly different between the original and the multivariant binary. Furthermore, no specific variant can be inferred from execution times gathered from the multivariant binary. MEWE contributes to mitigate potential attacks based on predictable execution times.

6 RELATED WORK

Our work is in the area of software diversification for security, a research field discovered by researchers Forrest [26] and Cohen [21]. We contribute a novel technique for multivariant execution, and discuss related work in Section 2. Here, we position our contribution with respect to previous work on randomization and security for WebAssembly.

6.1 Related Work on Randomization

A randomization technique creates a set of unique executions for the very same program [12]. Seminal works include instruction-set randomization [9, 34] to create a unique mapping between artificial CPU instructions and real ones. This makes it very hard for an attacker ignoring the key to inject executable code. Compiler-based techniques can randomly introduce NOP and padding to statically diversify programs. [30] have explored how to use NOP and it breaks the predictability of program execution, even mitigating certain exploits to an extent.

Chew and Song [19] target operating system randomization. They randomize the interface between the operating system and the user applications: the system call numbers, the library entry points (memory addresses) and the stack placement. All those techniques are dynamic, done at runtime using load-time preprocessing and rewriting. Bathkar et al. [12, 13] have proposed three kinds of randomization transformations: randomizing the base addresses of applications and libraries memory regions, random permutation of the order of variables and

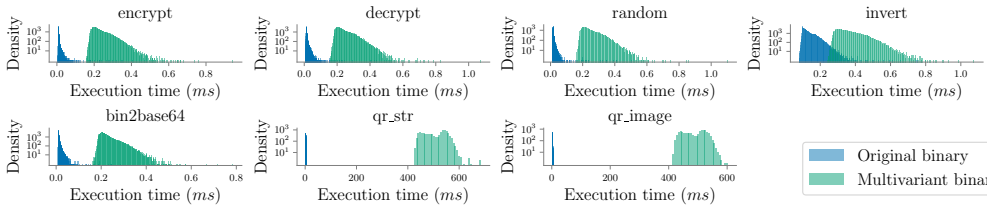


Fig. 4. Execution time distributions. Each subplot represents the distribution for a single endpoint, blue for the original endpoint and green for the multivariant binary. The X axis shows the execution time in milliseconds and the Y axis shows the density distribution in logarithmic scale.

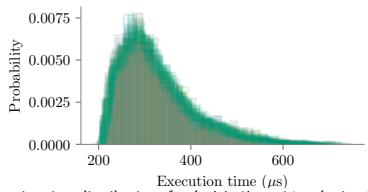


Fig. 5. Execution time distributions for the `bin2base64` endpoint. Each opaque curve represents an execution time distribution of a unique execution path out of the 41 observed.

routines, and the random introduction of random gaps between objects. Dynamic randomization can address different kinds of problems. In particular, it mitigates a large range of memory error exploits. Recent work in this field include stack layout randomization against data-oriented programming [7] and memory safety violations [37], as well as a technique to reduce the exposure time of persistent memory objects to increase the frequency of address randomization [60].

We contribute to the field of randomization, at two stages. First, we automatically generate variants of a given program, which have different WebAssembly code and still behave the same. Second, we randomly select which variant is executed at runtime, creating a multivariant execution scheme that randomizes the observable execution trace at each run of the program.

Davi et al. proposed Isomeron [25], an approach for execution-path randomization. Isomeron simultaneously loads the original program and a variant. While the program is running, Isomeron continuously flips a coin to decide which copy of the program should be executed next at the level of function calls. With this strategy, a potential attacker cannot predict whether the original or the variant of a program will execute. MEWE proposes two key novel contributions. First, our code diversification step can generate variants of complex control flow structures by inferring constants or loop unrolling. Second, MEWE interconnects hundreds of variants and several randomization dispatchers in a single binary, increasing by orders of magnitude the runtime uncertainty about what code will actually run, compared to the choice among 2 variants proposed by Isomeron.

6.2 Related work on WebAssembly Security

The reference piece about WebAssembly security is by Lehmann et al. [38], which presents three attack primitives. Lehmann et al. have then followed up with a large-scale empirical study of WebAssembly binaries [29]. Narayan et al. [45] remark that the security model of WebAssembly is vulnerable to Spectre attacks. This means that WebAssembly sandboxes may be hijacked and leak memory. They propose to modify the Lucet compiler used by Fastly to incorporate LLVM fence instructions[4] in the machine code generation, trying to avoid speculative execution mistakes. Johnson et al. [33], on the other hand, propose fault isolation for WebAssembly binaries, a technique that can be applied before being deployed to the edge-cloud platforms. Stievenart et al. [55] design a static analysis dedicated to information flow problems. Bian et al. [14] performs runtime monitoring of WebAssembly to detect cryptojacking. The main difference with our work is that our defense mechanism is larger in scope, meant to tackle "yet unknown" vulnerabilities. Notably, MEWE is agnostic from the last-step compilation pass that translates Wasm to machine code, which means that the multivariant binaries can be deployed on any edge-cloud platform that can receive WebAssembly endpoints, regardless of the underlying hardware.

7 CONCLUSION

In this work we propose a novel technique to automatically synthesize multivariant binaries to be deployed on edge computing platforms. Our tool, MEWE, operates on a single service implemented as a WebAssembly binary. It automatically generates functionally equivalent variants for each function that implements the service, and combines all the variants in a single WebAssembly binary, which exact execution path is randomized at runtime. Our evaluation with 7 real-world cryptography and QR encoding services shows that MEWE can generate hundreds of function variants and combine them into binaries that include from thousands to millions of possible execution paths. The deployment and execution of the multivariant binaries on the Fastly cloud platform showed that they actually exhibit a very high diversity of execution at runtime, in single edge nodes, as well as Internet scale.

Future work with MEWE will address the trade-off between a large space for execution path randomization and the computation cost of large-scale runtime randomization. In addition, the synthesis of a large pool of variants supports the exploration of the concurrent execution of multiple variants to detect misbehaviors in services deployed at the edge. Besides, several components of MEWE are implemented to operate at the level of the LLVM intermediate language. These components are compatible with other LLVM workflows. We plan to extend MEWE for other LLVM workflows, such as Rust, a popular workflow for Wasm applications and libraries.

REFERENCES

[1] 2020. Markdown to HTML. https://markdown-converter.edgecompute.app/

[4] https://llvm.org/doxygen/classllvm_1_1FenceInst.html

[2] 2020. The power of serverless, 72 times over. https://www.fastly.com/blog/the-power-of-serverless-at-the-edge

[3] 2021. Global CDN Disruption. https://status.fastly.com/incidents/vpk0ssybt3bj

[4] 2021. The New York Times on failure, risk, and prepping for the 2016 US presidential election – Fastly. https://www.fastly.com/blog/new-york-times-on-failure-risk-and-prepping-2016-us-presidential-election

[5] 2021. WebAssembly System Interface. https://github.com/WebAssembly/WASI

[6] Onur Acıiçmez, Werner Schindler, and Çetin K Koç. 2007. Cache based remote timing attack on the AES. In *Cryptographers' track at the RSA conference*. Springer, 271–286.

[7] Misiker Tadesse Aga and Todd Austin. 2019. Smokestack: thwarting DOP attacks with runtime stack layout randomization. In *Proc. of CGO*. 26–36. https://drive.google.com/file/d/12TvsrgL8Wt6lMfe6ASUp8y69L-bCVao0/view

[8] Simon Allier, Olivier Barais, Benoit Baudry, Johann Bourcier, Erwan Daubert, Franck Fleurey, Martin Monperrus, Hui Song, and Maxime Tricoire. 2015. Multitier diversification in Web-based software applications. *IEEE Software* 32, 1 (2015), 83–90. https://doi.org/10.1109/MS.2014.150

[9] Elena Gabriela Barrantes, David H Ackley, Stephanie Forrest, Trek S Palmer, Darko Stefanovic, and Dino Dai Zovi. 2003. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proc. CCS*. 281–289.

[10] Nicolas Belleville, Damien Couroussé, Karine Heydemann, and Henri-Pierre Charles. 2018. Automated Software Protection for the Masses Against Side-Channel Attacks. *ACM Trans. Archit. Code Optim.* 15, 4, Article 47 (nov 2018), 27 pages. https://doi.org/10.1145/3281662

[11] Daniel J Bernstein. 2005. Cache-timing attacks on AES. (2005).

[12] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. 2003. Address obfuscation: an efficient approach to combat a board range of memory error exploits. In *Proceedings of the USENIX Security Symposium*.

[13] Sandeep Bhatkar, Ron Sekar, and Daniel C DuVarney. 2005. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the USENIX Security Symposium*. 271–286.

[14] Weikang Bian, Wei Meng, and Mingxue Zhang. 2020. Minethrottle: Defending against wasm in-browser cryptojacking. In *Proceedings of The Web Conference 2020*. 3112–3118.

[15] Tegan Brennan, Nicolás Rosner, and Tevfik Bultan. 2020. JIT Leaks: inducing timing side channels through just-in-time compilation. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1207–1222.

[16] Danilo Bruschi, Lorenzo Cavallaro, and Andrea Lanzi. 2007. Diversified process replicæ for defeating memory error exploits. In *Proc. of the Int. Performance, Computing, and Communications Conference*.

[17] David Bryant. 2020. Webassembly outside the browser: A new foundation for pervasive computing. In *Proc. of ICWE 2020*. 9–12.

[18] Javier Cabrera-Arteaga, Orestis Floros Malivitsis, Oscar Vera-Pérez, Benoit Baudry, and Martin Monperrus. 2021. CROW: Code Diversification for WebAssembly. In *MADWeb, NDSS 2021*.

[19] Monica Chew and Dawn Song. 2002. *Mitigating buffer overflows by operating system randomization*. Technical Report CS-02-197. Carnegie Mellon University.

[20] Sharon Choy, Bernard Wong, Gwendal Simon, and Catherine Rosenberg. 2014. A hybrid edge-cloud architecture for reducing on-demand gaming latency. *Multimedia systems* 20, 5 (2014), 503–519.

[21] Frederick B Cohen. 1993. Operating system protection through program evolution. *Computers & Security* 12, 6 (1993), 565–584.

[22] Bart Coppens, Bjorn De Sutter, and Jonas Maebe. 2013. Feedback-driven binary code diversification. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 4 (2013), 1–26.

[23] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. 2006. N-variant systems: a secretless framework for security through diversity. In *Proc. of USENIX Security Symposium* (Vancouver, B.C., Canada) *(USENIX-SS'06)*. http://dl.acm.org/citation.cfm?id=1267336.1267344

[24] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. 2015. Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity. In *NDSS*. 8–11.

[25] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z Snow, and Fabian Monrose. 2015. Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming. In *NDSS*.

[26] Stephanie Forrest, Anil Somayaji, and David H Ackley. 1997. Building diverse computer systems. In *Proceedings. The Sixth Workshop on Hot Topics in Operating Systems*. IEEE, 67–72.

[27] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 185–200.

[28] Pat Hickey. 2018. *Announcing Lucet: Fastly's native WebAssembly compiler and runtime*. Technical Report. https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime

[29] Aaron Hilbig, Daniel Lehmann, and Michael Pradel. 2021. An Empirical Study of Real-World WebAssembly Binaries: Security, Languages, Use Cases. In *Proceedings of the Web Conference 2021*. 2696–2708.

[30] Todd Jackson. 2012. *On the Design, Implications, and Effects of Implementing Software Diversity for Security*. Ph.D. Dissertation. University of California, Irvine.

[31] Todd Jackson, Christian Wimmer, and Michael Franz. 2010. Multi-variant program execution for vulnerability detection and analysis. In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*. 1–4.

[32] Martin Jacobsson and Jonas Wåhslén. 2018. Virtual machine execution for wearables based on webassembly. In *EAI International Conference on Body Area Networks*. Springer, Cham, 381–389.

[33] Evan Johnson, David Thien, Yousef Alhessi, Shravan Narayan, Fraser Brown, Sorin Lerner, Tyler McMullen, Stefan Savage, and Deian Stefan. 2021. SFI safety for native-compiled Wasm. *NDSS. Internet Society* (2021).

[34] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. 2003. Countering code-injection attacks with instruction-set randomization. In *Proc. of CCS*. 272–280.

[35] Dohyeong Kim, Yonghwi Kwon, William N. Sumner, Xiangyu Zhang, and Dongyan Xu. 2015. Dual Execution for On the Fly Fine Grained Execution Comparison. *SIGPLAN Not.* (2015).

[36] Koen Koning, Herbert Bos, and Cristiano Giuffrida. 2016. Secure and efficient multi-variant execution using hardware-assisted process virtualization. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 431–442.

[37] Seongman Lee, Hyeonwoo Kang, Jinsoo Jang, and Brent Byunghoon Kang. 2021. SaVioR: Thwarting Stack-Based Memory Safety Violations by Randomizing Stack Layout. *IEEE Transactions on Dependable and Secure Computing* (2021). https://ieeexplore.ieee.org/iel7/8858/4358699/09369900.pdf

[38] Daniel Lehmann, Johannes Kinder, and Michael Pradel. 2020. Everything Old is New Again: Binary Security of WebAssembly. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association.

[39] Julian Lettner, Dokyung Song, Taemin Park, Per Larsen, Stijn Volckaert, and Michael Franz. 2018. PartiSan: fast and flexible sanitization via run-time partitioning. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 403–422.

[40] Hans Liljestrand, Thomas Nyman, Lachlan J Gunn, Jan-Erik Ekberg, and N Asokan. 2021. PACStack: an Authenticated Call Stack. In *30th USENIX Security Symposium (USENIX Security 21)*.

[41] Kangjie Lu, Meng Xu, Chengyu Song, Taesoo Kim, and Wenke Lee. 2018. Stopping memory disclosures via diversification and replicated execution. *IEEE Transactions on Dependable and Secure Computing* (2018).

[42] H. B. Mann and D. R. Whitney. 1947. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *Ann. Math. Statist.* 18, 1 (03 1947), 50–60. https://doi.org/10.1214/aoms/1177730491

[43] Matthew Maurer and David Brumley. 2012. TACHYON: Tandem execution for efficient live patch testing. In *21st USENIX Security Symposium (USENIX Security 12)*. 617–630.

[44] P. Mendki. 2020. Evaluating Webassembly Enabled Serverless Approach for Edge Computing. In *2020 IEEE Cloud Summit*. 161–166. https://doi.org/10.1109/IEEECloudSummit48914.2020.00031

[45] Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean Tullsen, et al. 2021. Swivel: Hardening WebAssembly against Spectre. In *USENIX Security Symposium*.

[46] Adam J O'Donnell and Harish Sethu. 2004. On achieving software diversity for improved network security using distributed coloring algorithms. In *Proceedings of the 11th ACM conference on Computer and communications security*. 121–131.

[47] Sebastian Österlund, Koen Koning, Pierre Olivier, Antonio Barbalace, Herbert Bos, and Cristiano Giuffrida. 2019. kMVX: Detecting kernel information leaks with multi-variant execution. In *ASPLOS*.

[48] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing digital side-channels through obfuscated execution. In *24th USENIX Security Symposium (USENIX Security 15)*. 431–446.

[49] Barbara G Ryder. 1979. Constructing the call graph of a program. *IEEE Transactions on Software Engineering* 3 (1979), 216–226.

[50] Babak Salamat, Andreas Gal, Todd Jackson, Karthik Manivannan, Gregor Wagner, and Michael Franz. 2007. *Stopping Buffer Overflow Attacks at Run-Time: Simultaneous Multi-Variant Program Execution on a Multicore Processor*. Technical Report. Technical Report 07-13, School of Information and Computer Sciences, UCIrvine.

[51] Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. 2009. Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proceedings of the 4th ACM European conference on Computer systems*. 33–46.

[52] Babak Salamat, Todd Jackson, Gregor Wagner, Christian Wimmer, and Michael Franz. 2011. Runtime Defense against Code Injection Attacks Using Replicated Execution. *IEEE Trans. Dependable Secur. Comput.* 8, 4 (2011), 588–601. https://doi.org/10.1109/TDSC.2011.18

[53] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight isolation for efficient stateful serverless computing. In *USENIX Annual Technical Conference*. 419–433.
[54] Natalie Silvanovich. 2018. *The Problems and Promise of WebAssembly*. Technical Report. https://googleprojectzero.blogspot.com/2018/08/the-problems-and-promise-of-webassembly.html
[55] Quentin Stiévenart and Coen De Roover. 2020. Compositional Information Flow Analysis for WebAssembly Programs. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 13–24.
[56] Tarik Taleb, Konstantinos Samdanis, Badr Mada, Hannu Flinck, Sunny Dutta, and Dario Sabella. 2017. On Multi-Access Edge Computing: A Survey of the Emerging 5G Network Edge Cloud Architecture and Orchestration. *IEEE Comm. Surveys & Tutorials* 19, 3 (2017).
[57] Kenton Varda. 2018. *WebAssembly on Cloudflare Workers*. Technical Report. https://blog.cloudflare.com/webassembly-on-cloudflare-workers/
[58] Stijn Volckaert, Bart Coppens, and Bjorn De Sutter. 2015. Cloning your gadgets: Complete ROP attack immunity with multi-variant execution. *IEEE Transactions on Dependable and Secure Computing* 13, 4 (2015).
[59] Alexios Voulimeneas, Dokyung Song, Per Larsen, Michael Franz, and Stijn Volckaert. 2021. dMVX: Secure and Efficient Multi-Variant Execution in a Distributed Setting. In *Proceedings of the 14th European Workshop on Systems Security*. 41–47.
[60] Yuanchao Xu, Yan Solihin, and Xipeng Shen. 2020. Merr: Improving security of persistent memory objects via efficient memory exposure reduction and randomization. In *Proc. of ASPLOS*. 987–1000. https://dl.acm.org/doi/pdf/10.1145/3373376.3378492

A DISPATCHER FUNCTION CODE

```
define internal i32 @b64_byte2urlsafe_char(i32 %0) {
    entry:
      %1 = call i32 @discriminate(i32 3)
      switch i32 %1, label %end [ i32 0, label %case_43_ i32 1, label
            %case_44_]
    case_43_: ; preds = %entry
      %2 = call i32 @b64_byte_to_urlsafe_char_43_(%0)
      ret i32 %2
    case_44_: ; preds = %entry
      %3 = <body of b64_byte_to_urlsafe_char_44_>
      ret i32 %3
    end: ; preds = %entry
      %4 = call i32 @b64_byte2urlsafe_char_original(%0)
      ret i32 %4
}
```

Listing 3. Dispatcher function embedded in the multivariant binary of the bin2base64 endpoint of libsodium, which corresponds to the rightmost green node in Figure 2.

B MULTIVARIANT BINARY EXECUTION AT THE EDGE

When a WebAssembly binary is deployed on an edge platform, it is translated to machine code on the fly. For our experiment, we deploy on the production edge nodes of Fastly. This edge computing platform uses Lucet, a native WebAssembly compiler and runtime, to compile and run the deployed Wasm binary [5]. Lucet generates x86 machine code and ensures that the generated machine code executes inside a secure sandbox, controlling memory isolation.

Figure 6 illustrates the runtime behavior of the original and the multivariant binary, when deployed on an Edge node. The top most diagram illustrates the execution trace for the original of the endpoint `bin2base64`. When the HTTP request with the input `"HelloWorld!"` is received, it invokes functions $f1$, $f2$ followed by 27 recursive calls of function $f3$. Then, the endpoint sends the result `"0x000xccv0x10x00b3Jsx130x000x00 0x00xpopAHRvdGE="` of its base64 encoding in an HTTP response.

[5] https://github.com/bytecodealliance/lucet

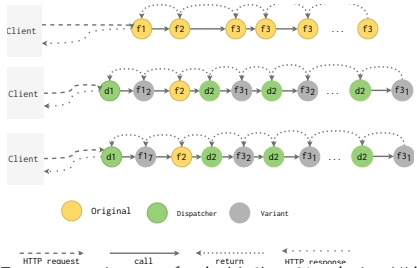


Fig. 6. Top: an execution trace for the `bin2base64` endpoint. Middle and bottom: two different execution traces for the multivariant `bin2base64`, exhibited by two different requests with exactly the same input.

The two diagrams at the bottom of Figure 6 illustrate two executions traces observed through two different requests to the endpoint `bin2base64`. In the first case, the request first triggers the invocation of dispatcher $d1$, which randomly decides to invoke the variant $f1_2$; then $f2$, which has not been diversified by MEWE, is invoked; then the recursive invocations to $f3$ are replaced by iterations over the execution of dispatcher $d2$ followed by a random choice of variants of $f3$. Eventually the result is computed and sent back as an HTTP response. The second execution trace of the multivariant binary shows the same sequence of dispatcher and function calls as the previous trace, and also shows that for a different requests, the variants of $f1$ and $f3$ are different.

The key insights from these figures are as follows. First, from a client's point of view, a request to the original or to a multivariant endpoint, is completely transparent. Clients send the same data, receive the same result, through the same protocol, in both cases. Second, this figure shows that, at runtime, the execution paths for the same endpoint are different from one execution to another, and that this randomization process results from multiple random choices among function variants, made through the execution of the endpoint.

C VARIANTS PRESERVATION

During our experiments, we checked for code diversity preservation after compilation. In this work, diversity is introduced through transformation on WebAssembly code, which is then compiled by the Lucet compiler. Compilation might perform some normalization and optimization passes when translating from WebAssembly to machine code. Thus, some variants synthesized by MEWE might not be preserved, i.e., Lucet could generate the same machine code for two WebAssembly variants. To assess this potential effect, we compare the level of code diversity among the WebAssembly variants and among the machine code variants produced by Lucet. This experiment reveals that the translation to machine code preserves a high ratio of function variants, i.e., approx 96% of the generated variants are preserved. This result also indicates that the machine code variants preserve the potential for large numbers of possible execution paths.

WEBASSEMBLY DIVERSIFICATION FOR MALWARE EVASION

Javier Cabrera-Arteaga, Tim Toady, Martin Monperrus, Benoit Baudry
Computers & Security, Volume 131, 2023

https://www.sciencedirect.com/science/article/pii/S0167404823002067

Computers & Security 131 (2023) 103296

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

WebAssembly diversification for malware evasion

Javier Cabrera-Arteaga*, Martin Monperrus, Tim Toady, Benoit Baudry

KTH Royal Institute of Technology, Stockholm, Sweden

ARTICLE INFO

Article history:
Received 21 December 2022
Revised 27 April 2023
Accepted 13 May 2023
Available online 18 May 2023

Keywords:
WebAssembly
Cryptojacking
Software diversification
Malware evasion

ABSTRACT

WebAssembly has become a crucial part of the modern web, offering a faster alternative to JavaScript in browsers. While boosting rich applications in browser, this technology is also very efficient to develop cryptojacking malware. This has triggered the development of several methods to detect cryptojacking malware. However, these defenses have not considered the possibility of attackers using evasion techniques. This paper explores how automatic binary diversification can support the evasion of WebAssembly cryptojacking detectors. We experiment with a dataset of 33 WebAssembly cryptojacking binaries and evaluate our evasion technique against two malware detectors: VirusTotal, a general-purpose detector, and MINOS, a WebAssembly-specific detector. Our results demonstrate that our technique can automatically generate variants of WebAssembly cryptojacking that evade the detectors in 90% of cases for VirusTotal and 100% for MINOS. Our results emphasize the importance of meta-antiviruses and diverse detection techniques and provide new insights into which WebAssembly code transformations are best suited for malware evasion. We also show that the variants introduce limited performance overhead, making binary diversification an effective technique for evasion.

1. Introduction

WebAssembly is a binary format that has become an essential component of the web. It first appeared in 2017 as a fast and safe complement for JavaScript (Haas et al., 2017). The language provides low-level constructs enabling efficient execution, much closer to native code than JavaScript. Since its inception, the adoption of WebAssembly has grown exponentially, even outside the web (Cabrera Arteaga et al., 2022). Its early adoption by malicious actors is further evidence of WebAssembly's success.

The primary black-hat usage of WebAssembly is cryptojacking (Rokicki et al., 2022). Such WebAssembly code mines cryptocurrencies on users' browsers for the benefit of malicious actors and without the consent of the users (Musch et al., 2019b). The main reason for this phenomenon is that the core foundation of cryptojacking is: the faster, the better. In this context, WebAssembly, a binary instruction format designed to be portable and fast, is a feasible technology for implementing and distributing cryptojacking over the web. A Kaspersky report about the state of cryptojacking in the first three quarters of 2022 confirms the steady growth in the usage of cryptominers (Kaspersky, 2022). The report shows that Monero (2022) is the most used cryptocurrency for cryptomining in the browser. Attackers might hide WebAssembly cryptominers (Tekiner et al., 2021) in multiple locations inside web applications.

Antivirus and browsers provide support for detecting cryptojacking. For example, the Firefox browser supports the detection of cryptomining by using deny lists (Mozilla, 2019). The academic community also provides related work on detecting or preventing WebAssembly cryptojacking (Bian et al., 2020; Kelton et al., 2020; Kharraz et al., 2019; Naseem et al., 2021; Romano et al., 2020; Wang et al., 2018). Yet, it is known that black-hats can use evasion techniques to bypass detection. Only the previous work of Bhansali et al. (2022) investigates the possibility of WebAssembly cryptojacking to evade detection techniques. This is a crucial motivation for our work, one of the first to study WebAssembly malware evasion.

Our work is different from Bhansali et al. (2022)'s in the following aspects. First, we extend the evaluation of MINOS by using VirusTotal and, we empirically demonstrate that this latter is a valid cryptojacking malware meta-detector for WebAssembly to be used as baseline (see Section 5). Second, we conduct an evaluation of the correctness and efficiency of the Wasm variants, which provides insights into the trade-offs and limitations of bytecode-level transformations for malware evasion in WebAssembly. Our approach, performs bytecode transformations at the WebAssembly level, instead of source code based transformations like the

* Corresponding author.
E-mail addresses: javierca@kth.se (J. Cabrera-Arteaga), monperrus@kth.se (M. Monperrus), toady@eecs.kth.se (T. Toady), baudry@kth.se (B. Baudry).

https://doi.org/10.1016/j.cose.2023.103296

Bhansali's technique. We focus on software diversification techniques in the spirit of Cohen (1993), as this technique has the ability to generate many variants, while the impact on the binary size and performance can be controlled through the selection of specific diversification transformations (Lundquist et al., 2016).

In this paper, we design, implement and evaluate a full-fledged evasion pipeline for WebAssembly. Concretely, we use wasm-mutate as a diversifier (Bytecodealliance, 2021), which implements 135 possible bytecode transformations, grouped into three categories: peephole operator, module structure, and control flow. We demonstrate the effectiveness of our evasion technique against two cryptojacking detectors: VirusTotal, a general detection tool that comprises 60 antiviruses and, MINOS (Naseem et al., 2021), a WebAssembly-specific detector.

We evaluate our proposed evasion technique on 33 cryptojacking malware that we curated from the 8643 binaries of the wasm-bench dataset (Hilbig et al., 2021), to our knowledge, the most exhaustive collection of real-world WebAssembly binaries. We experiment with the 33 binaries marked as potentially dangerous by at least one antivirus vendor of VirusTotal. We empirically demonstrate that evasion is possible for all of these 33 real-world WebAssembly cryptojacking malware while using a WebAssembly-specific detector. Remarkably, we find 30 cryptominers for which our technique successfully generates variants that evade VirusTotal. Our set of malware includes 6 cryptojacking programs that are fully reproducible in a controlled environment. With them, we assess that our evasion method does not affect malware correctness and generates fully functional malware variants with minimal overhead.

Our work provides evidence that the malware detection community has opportunities to strengthen the automatic detection of cryptojacking WebAssembly malware. The results of this work are actionable, as we provide quantitative evidence on specific malware transformations on which detection methods can focus. To sum up, the contributions of this work are:

- A full-fledged cryptojacking malware evasion pipeline for WebAssembly, based on a state-of-the-art binary diversification. We provide the repository of the tool at https://github.com/ASSERT-KTH/wasm_evasion.
- A systematic evaluation of our cryptojacking evasion pipeline, including effectiveness, performance, and correctness.
- Actionable evidence on which transformations are better for evading WebAssembly cryptojacking detectors, calling for future work from the academic and the industrial community alike.
- A reproducible comparison of VirusTotal with MINOS, a WebAssembly-specific detector, showing the relevance of VirusTotal as a valid and practical cryptojacking meta-detector.

This paper is structured as follows. In Section 2, we introduce WebAssembly for cryptomining and the state-of-the-art on malware detection and evasion techniques and its limitations for WebAssembly cryptojacking. In Section 3, we instantiate and explain malware evasion for WebAssembly cryptojacking in a real scenario. We follow with the technical description of our malware evasion algorithms in Section 4. We formulate our research questions in Section 5, answering them in Section 6. We discuss our research in Section 7, in order to help future research projects on similar topics. We finalize with our conclusions Section 8.

2. Background & related work

In this section, we introduce WebAssembly. Besides, we illustrate its usage for cryptojacking. Then, we discuss how WebAssembly cryptojacking can be detected, and the most common techniques used to evade such detection.

2.1. WebAssembly

WebAssembly (Wasm) is a binary instruction set meant initially for the web. It was adopted as a standard language for the web by the W3C in 2017, building upon the work of Haas et al. (2017). One of Wasm's primary advantages is that it defines its own Instruction Set Architecture (ISA), which is both straightforward and platform-independent. As a result, a Wasm binary can execute on virtually any platform, including web browsers and server-side environments. Since its introduction, all major web browsers have implemented support for WebAssembly, reporting to be only 10% slower than machine code during runtime.

WebAssembly programs are compiled ahead-of-time from source languages such as C/C++, Rust, and Go, utilizing compilation pipelines like LLVM. This allows Wasm to benefit from ahead-of-time compiling optimizations, improving its performance. A Wasm binary is comprised of sections, which are consecutive sequences of bytes in the binary file. In contrast to the absolute order of sections in Windows Portable Programs, sections in Wasm binaries have a relative order between them. Thus, Wasm can be considered a more flexible binary format.

WebAssembly programs operate on a virtual stack that allows for only four data types: i32, i64, f32, and f64. These same data types are used to annotate the numeric operations in the WebAssembly code. Additionally, a WebAssembly program might include several custom sections. For example, binary producers such as compilers use it to store metadata. A WebAssembly code also declares memories and globals, which are used to store, manipulate and share data during program execution, e.g. to share data with the host engine of the WebAssembly binary.

WebAssembly is designed with isolation as a primary consideration. For instance, a WebAssembly binary cannot access the memory of other binaries or interact directly with a browser's built-in API, such as the DOM or the network. Instead, communication with these features is constrained to functions imported from the host engine, ensuring a secure and safe Wasm environment. Moreover, control flow in WebAssembly is managed through explicit labels and well-defined blocks, which means that jumps in the program can only occur inside blocks, unlike regular assembly code.

In Listing 1, we provide an example of a C program that contains a function declaration, a loop, a loop conditional, and a memory access. When the C code is compiled to WebAssembly, it produces the code shown in Listing 2. The stack operations are folded with parentheses. The module in the example contains the components described previously.

2.2. Malware in WebAssembly

The use cases of WebAssembly in browsers focuses on computation-intensive activities such as gaming or image processing. Also, malign actors have taken advantage of WebAssembly to carry out their activities, and cryptojacking is the most common usage observed so far (Musch et al., 2019a; 2019b). The reason for this is that cryptojacking involves executing vast amounts of hash

```
int a[100];
void cc1(){
    for(int i = 0; i < 100;
        ↪ i++){
        a[i] = i;
    }
}
```

Listing 1. C program containing function declaration, loops, conditionals and memory access.

```
(module
  (@custom "producer" "llvm.." )
  (func (;5;) (type 1)
    (loop  ;; label = @1
      (if  ;; label = @2
        (i32.eqz
          (i32.ge_s
            (i32.load
              (local.get 0))
            (i32.const 100)))
        (then (i32.store
            (i32.add
              (i32.shl
                (i32.load
                  (local.get 0))
                (i32.const 2))
              (i32.const 1024))
            (i32.load
              (local.get 0)))
          (i32.store
            (local.get 0)
            (i32.add
              (i32.load
                (local.get 0))
              (i32.const 1)))
          (br 1 (;@1;)))))))
  )
  (memory (;0;) 256)
  (global (;1;) i32 (i32.const 0))
  (export "memory" (memory 0)))
```

Listing 2. WebAssembly code for C code in Listing 1.

functions, which requires significant computing resources. In comparison to JavaScript, WebAssembly is significantly faster at handling these intense hashing operations in the browser (Haas et al., 2017).

Web cryptojacking is often carried out by including a malicious JavaScript+WebAssembly payload, which then execute on the victim's browser without their knowledge (Tekiner et al., 2021). For example, websites that offer illegal download or adult sites, often include cryptojacking in their webpages to generate passive income. Since cryptojacking is difficult to detect and remove, it can remain on a victim's computer for an extended period, continuing to consume resources and to generate income for the attacker. This lucrative form of malware does need vulnerabilities or stealing credentials.

2.3. Malware detection

Malware detection determines if a binary is malicious or not. This process can be based on static, dynamic, or hybrid analysis (Aslan and Samet, 2020). In this section, we highlight works in the area of malware detection. Static-based approaches analyze the source code or the binary to find malign patterns without executing them. The literature reports a range of techniques, from simple checksum checking to advanced machine learning methods, that have subsequently been adopted by commercial antiviruses (Botacin et al., 2022; Li et al., 2021). In the context of WebAssembly, MineSweeper is a detection method based on static analysis (Konoth et al., 2018) of WebAssembly. Its detection strategy depends on the knowledge of the internals of CryptoNight, one popular library for cryptomining. In the same context, MINOS is a state-of-art static detection tool that converts WebAssembly binaries to vectors for malware detection (Naseem et al., 2021).

MINOS is a practical approach for detecting malicious WebAssembly binaries. It works by converting the Wasm binary's bytestream into a 100×100 grayscale image, which is then fed into a Convolutional Neural Network (CNN). The CNN has learned patterns in the image to classify it as either benign or malicious. This approach is similar to image-based methods used in other areas (Liu and Wang, 2016), e.g., for detecting Windows malware (Kalash et al., 2018; Lachtar et al., 2023). We believe MINOS is the optimal static approach to detect WebAssembly malware due to its simplicity and practicality, such as being easily implemented as a browser extension.

Dynamic analysis for malware detection is based on the execution of the malware code to identify potentially dangerous behaviors (Egele et al., 2008). Usually, this is done by monitoring some functions, such as API calls. For example, BLADE (Lu et al., 2010) is a Windows kernel extension that aims to eliminate drive-by malware installations. It wraps the filesystem for browser downloads for which user consent has been involved. It thwarts the ability of browser-based exploits to download and execute malicious content surreptitiously. SEISMIC and MineThrotle also perform dynamic analyses (Bian et al., 2020; Wang et al., 2018) on WebAssembly binaries to profile instructions that are specific to cryptominers. For example, cryptominers overly execute `XOR` instructions. SEISMIC and MineThrotle use machine learning approaches to classify the binary as benign or malign based on collecting runtime profiles. On the same topic, MinerRay (Romano et al., 2020) detects cryptojacking in WebAssembly binaries by analyzing their control flow graph at runtime, searching for structures that are characteristic of encryption algorithms commonly used for cryptojacking. CoinSpy is another malware detector based on dynamic analysis (Kelton et al., 2020). It uses a convolutional neural network to analyse the computation, network, and memory information caused by cryptojackers running in client browsers.

Hybrid approaches use a mix of static and dynamic detection techniques. The main reason to use hybrid approaches is the impracticability of executing the whole program. Thus, only pieces of code that can be quickly executed are dynamically analyzed. For example, AppAudit embodies a novel dynamic analysis for Android applications that can simulate the execution of some parts of the program (Xia et al., 2015). For WebAssembly, Outguard (Kharraz et al., 2019) trains a Support Vector Machine model with a combination of cryptomining function names obtained statically and dynamic information such as the number of web workers used in the web application that is analyzed.

It is possible to combine several independent detectors into a meta-antivirus. Each detector embeds some heuristics that are good at detecting specific types of malware (Moser et al., 2007). Hence, their combination can effectively detect a broader range of malware, e.g., using relationship analysis. VirusTotal (GoogleLLC, 2022; VirusTotal, 2020) is a consolidated meta-antivirus. VirusTotal operates with 60 antivirus vendors to provide malware scanning. Through its API, a program can be labeled by 60 antiviruses. This aggregation is used to determine if an asset under analysis is malicious, e.g., by voting. Previous works used VirusTotal to assess detection efficiency, Peng et al. (2019) because it is a proxy to evaluate state-of-art techniques in combination with commercial antiviruses. In this work, we follow the same methodology, using VirusTotal to assess our technique's ability to evade cryptojacking detectors.

While concerns have been raised about the use of VirusTotal for some malware and file type families (Botacin et al., 2020), it can be considered for WebAssembly cryptojacking detection. We empiri-

cally highlight later in this paper that VirusTotal is slightly better than the WebAssembly-specific detector MINOS regarding the detection of cryptojacking malware.

2.4. Malware evasion

Malware evasion techniques aim at avoiding malware detection (Afianian et al., 2019) Potential attackers use a wide range of techniques to achieve evasion, such as genetic programming (Castro et al., 2019). With time, the techniques to avoid detection have grown in complexity and sophistication (Aghakhani et al., 2020). For example, Chua and Balachandran (2018) proposed a framework to automatically obfuscate Android applications' source code using method overloading, opaque predicates, try-catch, and switch statement obfuscation, creating several versions of the same malware. Also, machine learning approaches have been used to create evading malware (Dasgupta and Osman, 2021), based on a corpus of pre-existing malware (Bostani and Moonsamy, 2021). While most approaches try to break static malware detectors, more sophisticated techniques avoid dynamic detection, usually involving throttling techniques or dynamic anti-analysis checks (Lu and Debray, 2013; Payer, 2014). Wang proposes the concept of Accrued Malicious Magnitude (AMM) to identify which malware features should be manipulated to maximize the likelihood of evading detection (Wang et al., 2021).

In the context of WebAssembly, malware evasion is nearly unexplored. Only Romano et al. (2022) recently proposed wobfuscator, a code obfuscation technique that transforms JavaScript code into a new JavaScript file and a set of WebAssembly binaries. Their technique mostly focuses on JavaScript evasion and not WebAssembly evasion.

Bhansali et al. (2022) propose a technique where WebAssembly binaries are transformed while maintaining the functionality with seven different source code obfuscation techniques. They evaluate the effectiveness of the techniques against MINOS (Naseem et al., 2021). They show these transformations can generate malware variants that evade the MINOS classifier.

3. WebAssembly cryptojacking malware evasion in practice

Figure 1 illustrates our attack scenario: a practical WebAssembly cryptojacking attack consists of three components: a WebAssembly binary, a JavaScript wrapper, and a backend cryptominer pool. The WebAssembly binary is responsible for executing the hash calculations, which consume significant computational resources. The JavaScript wrapper facilitates the communication between the WebAssembly binary and the cryptominer pool. Overall, a successful cryptojacking attack on a victim's browser consists in the following sequence of steps. First, the victim visits a web page infected with the cryptojacking code. The web page establishes a channel to the cryptominer pool, which then assigns a hashing job to the infected browser. The WebAssembly cryptominer calculates thousands of hashes inside the browser, in parallel using multiple browser workers (Mozilla, 2022). Once the malware server receives acceptable hashes, it is rewarded with cryptocurrencies for the mining. Then, the server assigns a new job, and the mining process starts over.

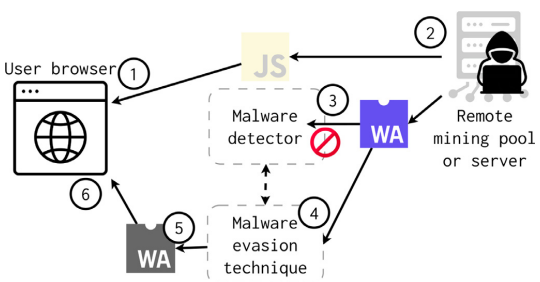


Fig. 1. WebAssembly evasion in practice. The user visits a webpage containing cryptojacking malware that uses network resources to operate. A malware detector blocks identified malicious WebAssembly binaries. The attacker, using a malware oracle, creates a WebAssembly cryptojacking malware variant that evades detection. Finally, the attacker delivers the modified binary, initiating the cryptojacking process and compromising the browser.

Some detection techniques discussed in Section 2.3 can be deployed in the browser directly to prevent cryptojacking. The primary objective of our work is to demonstrate the possibility of using code diversification to bypass cryptojacking defenses. Concretely, the following workflow can happen to successfully evade placed defenses: i) The user visits a webpage that contains a cryptojacking malware, which utilizes network resources to execute, (1) and (2) in Fig. 1. Cryptojacking malware can be injected through malicious browser extensions, malvertising, compromised websites, or deceptive links (Tekiner et al., 2021). ii) A malware detector blocks WebAssembly binaries that are identified as malicious (3). The malware detector system can be implemented locally or remotely. For instance, a proxy can intercept and send network resources to an external detector through the detector's API. iii) The attacker, based on a malware oracle, crafts a WebAssembly cryptojacking malware variant that evade the detection (4). iv) The attacker delivers the modified binary instead of the original one (5), which initiates the cryptojacking process and compromises the browser (6).

The idea is that attackers rapidly diversify their WebAssembly code to stay ahead of the defense system and maintain successful cryptojacking operations. Crucially, attackers must ensure that the diversified binaries they use for cryptojacking meet specific performance requirements, which is an aspect we will study in Section 4.

4. Diversification for malware evasion in WebAssembly

In this section, we explain a technique for potential attackers to craft a WebAssembly binary that evades detection (steps 4, 5 and 6 in Fig. 1).

In Fig. 2 we illustrate our generic architecture for the malware evasion component. The workflow starts by passing a WebAssembly malware binary to a software diversifier (1). The diversifier generates binary variants, which are passed to a malware oracle (2). The oracle returns labeling feedback for the binary variant: malware or benignware. The oracle result is the input for a fitness function that steers the construction of a new binary on top of the previously diversified one. This process is repeated until the malware oracle marks the mutated binary as benign or a timeout is reached (4). For the sake of open science and for fostering research on this important topic, our implementation is made publicly available on GitHub: https://github.com/ASSERT-KTH/wasm_evasion.

4.1. Diversifier

Conceptually, our approach is parametrized by a semantic preserving diversifier (Cohen, 1993). For our prototype implementation, we select one diversifier that supports wasm-to-wasm diversification and performs almost non-costly transformations: wasm-mutate (Bytecodealliance, 2021). This tool takes a WebAssembly module as input and returns a set of variants of that module. wasm-mutate follows the notion of program equivalence modulo input (Le et al., 2014), i.e., the variants should provide the same

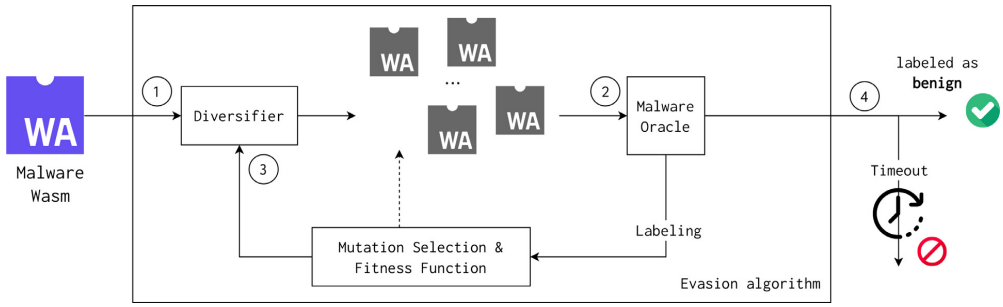


Fig. 2. Our original workflow of binary diversification for malware evasion in WebAssembly. The workflow begins with a WebAssembly malware binary sent to a software diversifier. The diversifier creates binary variants, which are analyzed by a malware oracle that labels them as malware or benignware. The oracle's results guide the development of a new binary based on the previous one. This process continues until the malware oracle labels the mutated binary as benign or a timeout occurs.

output for the same program inputs. It represents the search space for new variants as an e-graph (Willsey et al., 2020), and it exploits the property that any traversal through the e-graph represents a semantically equivalent variant of the input program. wasm-mutate can generate thousands of semantically equivalent variants from a single binary in a few minutes.

Wasm-mutate defines 135 possible transformations on an input WebAssembly binary, grouped into three categories. The peephole operator is the first category. It is responsible for rewriting instruction sequences in function bodies. It selects a random instruction within the binary functions and applies one or more of the 125 rewrite rules. Finally, it re-encodes the WebAssembly module with the new, rewritten expression to produce a binary variant. The second category of transformations in wasm-mutate implements module structure transformations. It operates at the level of the WebAssembly binary structure. It includes the following eight transformations: add a new type definition, add/modify custom sections, add a new function, add a new export, add a new import, add a new global, remove a type and remove a function. The third transformation category is on the control flow graph of a function code level. wasm-mutate performs two possible transformations: unroll a loop or swap the branches of a conditional block.

The decision of using wasm-mutate as a diversifier is based on three key factors. First, while diversification approaches for WebAssembly from LLVM sources exist (Cabrera Arteaga et al., 2022; Cabrera-Arteaga et al., 2021), compiler-based diversification may include compiler fingerprints in the built binaries, which is bad for stealthy evasion. Second, while optimization-based approaches could be used to diversify WebAssembly binaries from source code (Ren et al., 2021), optimizations are usually all applied at once, providing a smaller diversification space, hence fewer opportunities for evasion. Finally, wasm-mutate is a tool that implements many useful semantically equivalent transformations, making it well-suited as a diversifier with minimal engineering effort. Therefore, using wasm-mutate as a diversifier provides a practical approach for evasion in WebAssembly. We believe that attackers would take the same path and reach the same conclusion.

For the sake of illustration, in Listing 3, we present a variant of the WebAssembly code shown in Listing 2. We generate this variant using the wasm-mutate diversifier, with two transformations. The changes made to the original code are highlighted in orange and green. The first transformation, highlighted in orange, involves removing the custom section that indicates the producer of the original binary. The second transformation, highlighted in green, replaces the shift-left operation in the original binary with two consecutive multiplications of the same value.

```
(module
 - (@custom "producer" "llvm.." )
  (func (;5;) (type 1)
    (loop  ;; label = @1
      (if  ;; label = @2
        (i32.eqz
          (i32.ge_s
            (i32.load
              (local.get 0))
            (i32.const 100)))


        (then (i32.store
            (i32.add
              (i32.mul
                (i32.load
                  (local.get 0))
                (i32.load
                  (local.get 0))
              )
              (i32.const 1024)
            )
            (i32.load
              (local.get 0)))
          (i32.store
            (local.get 0)
            (i32.add
              (i32.load
                (local.get 0))
              (i32.const 1)))
          (br 1 (;@1;)))))
  )
  ...
```

Listing 3. Wasm-mutate transformation applied over Listing 2.

4.2. Malware oracle

To determine whether a given sample is malicious or not, we rely on a malware oracle. The simplest form of malware oracle is a binary classifier that outputs a label of {malware, benignware}. In addition to binary classifiers, we also consider numerical oracles

that provide a value representing the likelihood that a sample is malicious.

In our experiments, we use VirusTotal (2020) as our malware oracle. VirusTotal operates with 60 antivirus vendors to provide malware scanning. Users submit binaries, and they receive labels from different vendors. The resulting 60 labels can then be used to determine if a queried asset is malicious, e.g., by voting. VirusTotal can be used as both a binary and a numerical oracle (Zhu et al., 2020). We use VirusTotal as a binary oracle by returning malware if at least one vendor classifies the binary as malware. We also use VirusTotal as a numerical oracle, using the number (between 0 and 60) of oracles labeling a binary as malware.

Research (Botacin et al., 2020), classification labels assigned to samples by vendors in VirusTotal can change over time due to new antivirus releases. In our work, we operate under the assumptions outlined in Section 3 and consider a scenario where an attacker develops and executes an evasion technique in under an hour. This timeframe is significantly shorter than the time it takes for classification labels to change in VirusTotal, which typically takes several days.

4.3. Transformation selection & fitness function

The third step of our workflow consists of two actions towards synthesizing a malware variant: select a wasm-mutate transformation to apply; and determine if the transformation is applied. This latter decision depends on: 1) the result of the malware oracle at the previous iteration and 2) an estimation of the ability of the new transformation to generate an evading binary. For this estimation, we implement two variations of our evasion algorithm.

First, we use a binary malware oracle. In this context, we always apply the transformation. This is the *baseline evasion algorithm*, which is discussed in more detail later.

The second variation of the evasion algorithm includes a fitness function that uses a numerical oracle to estimate if the transformation should be applied. The fitness function uses the information from VirusTotal and is the total number of vendors (between 0 and 60) that label a binary as malware:

$$FF(m) = \Sigma_{i=0}^{i=60} \begin{cases} 1 \text{ if } v_i(m) \text{ returns } \texttt{malware} \\ 0 \text{ i.o.c} \end{cases}$$

This fitness function is used in our second proposed algorithm and is discussed in details below.

4.3.1. Baseline evasion

The baseline evasion algorithm is described in Algorithm 1. It uses VirusTotal as a binary oracle (true if at least one VirusTotal detects malware, false otherwise). In this algorithm, step 1 is in line 3, steps 2 and 4 are in lines 4 to 6, and step 3 is in line 7. Each iteration of this algorithm, "stacks" a transformation on top of the previous ones (line 7) until the binary is marked as benign (line 5) or the maximum number of iterations is reached (line 2).

With this algorithm, a transformation is randomly selected at each iteration, and always applied. Hence, the baseline algorithm can require many iterations and oracle queries to turn the original malware into a misclassified binary. Second, some transformations might suppress the effect of previous ones. Third, the baseline algorithm considers each vendor equally good at detecting a malware, which is naive as the vendors display considerable diversity regarding detection strength. An algorithm that would target evasion on the strongest vendor first would increase the overall performance of the evasion process. Finally, we might generate a binary that entirely evades the oracle but is unpractical in terms of size or its execution performance.

Algorithm 1: Baseline evasion algorithm.

```
input : binary W, diversifier D, Malware Oracle MO
output: Benign binary M'
M ← W while Not max iterations do
    M' ← D(M) if MO(M') == "benign" then
        return M';
    else
        M ← M';
return "Not evaded";
```

Algorithm 2: MCMC evasion algorithm.

```
input : binary W, diversifier D, fitness function FF
output: Benign binary M'
M ← W previous_fitness = FF(W) while Not max iterations do
    M' ← D(M) current_fitness = FF(M') if current_fitness == 0
    then
        // Zero means that none vendor marks// the binary as
        malware return M';
    else
        p ← random() if
        p < min(1, exp(σ previous_fitness / current_fitness)) then
            M ← M'; previous_fitness = current_fitness;
return "Not evaded";
```

4.3.2. MCMC evasion

To overcome the limitations of the baseline algorithm, we devise the *MCMC evasion algorithm*, which we now discuss. It is a Markov Chain Monte Carlo (MCMC) sampling (Hastings, 1970) of the transformations to apply (Schkufza et al., 2012). MCMC is used to sample from the space of transformation sequences to maximize the likelihood of oracle evasion in two ways.

The algorithm for the MCMC evasion is given in Algorithm 2. The halting condition is met when a mutated binary is marked as benign, or a maximum number of iterations is reached. The algorithm implements the MCMC in lines 6 to 15. The Markov decision function in line 12 is used to determine whether it is worth applying the transformation at this step or whether it should be skipped. This decision function is based on the current transformation's fitness and the fitness value saved at the previous step (line 14). The core idea is to favor a binary variant that evades the largest number of vendors (lines 4 and 13). Therefore, the number of oracle calls should decrease as the algorithm searches for transformations that converge toward total evasion. On the other hand, if a new transformation step decreases the fitness value, it is likely to be ignored. The classical MCMC acceptance criteria in line 12 is meant to prevent the algorithm from being stuck in local minima.

In line 12, the fraction calculated in the exponentiation is controlled by the σ parameter. By setting a low σ parameter, we can turn the MCMC evasion algorithm into a greedy algorithm. In this case, the algorithm selects a new transformation only if the fitness value is higher than in the previous iteration. On the contrary, if the σ parameter is significant, the algorithm searches for local maxima. In our experiments (Section 5), we explain how we select the values of σ.

The MCMC evasion algorithm addresses the three limitations of the baseline mentioned in the previous section. First, the fitness function selects transformations, thus reducing the total number of transformations that are actually performed. Second, MCMC aims at increasing fitness, which reduces the risk of suppressing a valuable transformation performed in the previous step. Third, by

favoring solutions that maximize the number of evaded vendors, MCMC biases the search towards evading the strongest vendors.

5. Experimental methodology

In this section, we enunciate the research questions around which we assess the ability of our technique to evade malware detectors. We also describe our dataset of malware, as well as the metrics we define to answer our research questions.

5.1. Research questions

- **RQ1. To what extent can cryptojacking malware detection be bypassed by WebAssembly diversification?** With this research question, we evaluate the feasibility of binary transformations on malware and how they affect the detection of cryptojacking.
- **RQ2. To what extent can the attacker minimize the number of calls to the cryptojacking detection oracle?** In real-world scenarios, the number of calls to the oracle is limited. With this question, we analyze the ability of our technique at limiting the number of oracles calls made during the evasion process.
- **RQ3. To what extent do the evasion techniques impact cryptojacking malware functionality?** The evasion algorithms might generate variants that evade the detectors but modify their core malicious functionality. This research question evaluates the correctness of the created variants, as well as their efficiency.
- **RQ4. What are the most effective transformations for WebAssembly cryptojacking malware evasion?** This research question provides empirical evidence on which types of transformations are better for WebAssembly cryptojacking evasion.
- **RQ5. To what extent can Wasm diversification evade cryptojacking detection with MINOS?** In this research question, we evaluate the feasibility of our technique for evading a state-of-the-art detector, MINOS, which is tailored to the analysis of WebAssembly.

5.2. Dataset selection

To answer our research questions, we curate a dataset of WebAssembly malware. For this, we filter the wasmbench dataset of Hilbig et al. (2021) to collect suspicious malware according to VirusTotal. The wasmbench dataset contains 8643 binaries collected from GitHub repositories and web pages in 2021. To our knowledge, this dataset is the newest and most exhaustive collection of real-world WebAssembly binaries. On August 2022, we passed the 8643 binaries of wasmbench to VirusTotal (2020), and we 33 binaries were marked as potentially dangerous by at least one antivirus vendor of VirusTotal. All malware were marked as cryptojacking programs and we use these WebAssembly binaries to answer our research questions for cryptojakcing malware evasion.

In Table 1 we describe the 33 binaries detected as malware by VirusTotal. The table contains the following properties as columns:

Table 1
The 33 real-world WebAssembly cryptojacking used in our experiments. The table contains: the 256 hash of the WebAssembly cryptojacking, its size in bytes, the number of instructions, the number of functions defined inside the binary and the number of VirusTotal vendors that detect the binary at the time of writing. The last column contains the origin of the binary.

Hash	S	#I.	#F.	#D	Origin
9d30e7f0	68,796	30,768	61	30	http archive
8ebf4e44	68,803	30,768	61	26	Web crawling
47d29959	68,796	30,768	61	31	Yara[a]
aafff587	97,551	47,033	72	6	SEISMIC[b]
dc11d82d	67,496	30,246	49	20	MinerRay[c]
0d996462	70,972	30,531	30	19	SEISMIC, MinerRay
fbdd1efa	94,270	45,905	40	18	SEISMIC
a32a6f4b	94,461	45,940	40	18	SEISMIC
d2141ff2	70,111	31,783	30	9	MinerRay, SEISMIC
046dc081	74,099	31,783	29	6	MinerRay, SEISMIC
24aae13a	62,458	28,339	37	4	SEISMIC
000415b2	62,466	28,339	37	3	SEISMIC
643116ff	73,010	31,866		6	MinerRay
006b2fb6	87,502	39,544	90	4	DeepMiner[d]
15b86a25	100,755	45,881	79	4	MinerRay
4cbdbbb1	104,666	47,916	62	3	SEISMIC
119c53eb	137,320	67,069	79	2	SEISMIC
f0b24409	77,572	34,918	59	2	MinerRay
c1be4071	77,572	34,918	59	2	MinerRay
a74a7cb8	77,572	34,918	59	2	MinerRay
a27b45ef	77,572	34,918	59	2	MinerRay
6b8c7899	77,572	34,918	59	2	MinerRay
68ca7c0e	77,572	34,918	59	2	MinerRay
65debcbe	77,572	34,918	59	2	MinerRay
5bc53343	77,572	34,918	59	2	MinerRay
59955b4c	77,572	34,918	59	2	MinerRay
942be4f7	103,520	46,208	79	4	MinerRay
fb15929f	77,054	33,562	112	4	MinerRay
7c36f462	121,931	55,839	79	4	MinerRay
89a3645c	75,003	34,134	58	2	MinerRay
dceaf65b	77,575	34,901	58	2	MinerRay
089dd312	79,883	34,989	58	2	MinerRay
e09c32c5	71,955	32,416	46	1	MinerRay

[a] Yara project https://github.com/davbo/yara-rs/tree/master/sample-miners.
[b] All Wasm binaries of the MinerRay project could be found at https://github.com/miner-ray/miner-ray.github.io/tree/master/Data/SampleWasmFiles.
[c] All Wasm binaries of the SEISMIC project could be found at https://github.com/wenhao1006/SEISMIC.
[d] Deepminer project https://github.com/deepwn/deepMiner.

the identifier of the WebAssembly binary which is the sha256 hash of its bytestream, its size in bytes, the number of instructions, the number of functions defined inside the binary and the number of VirusTotal vendors that detect the binary. The last column contains the origin of the binary according to the wasmbench dataset.[1]

The programs include between 30 and 70 functions, for a total number of instructions ranging from 30,531 to 55,839. The size of the programs ranges from 62 to 103 kilobytes. These binaries are detected as malicious by at least 1 antivirus, and at most 31. We have observed that 6 out of 33 binaries can be executed end-to-end.

To validate that the detected binaries are cryptojacking, we manually analyze each of the 33 binaries identified as malign. First, we observe that the binaries in the dataset originate from two primary sources: project SEISMIC and project MinerRay. SEISMIC (Wang et al., 2018) is a research project about instrumentation and monitoring at runtime to detect cryptojacking binaries. MinerRay is also a research project to detect crypto mining processes in web browsers (Romano et al., 2020). Both projects have collected the binaries as real cryptojacking from the web and the dataset is a union of them.

Second, we observee that all binaries share code from cryptonight (XMRIG, 2016), which is a library for cryptomining hashing. This observation is consistent with the findings of Romano et al. (2020). We find 5 binaries that are multivariant packages (Cabrera Arteaga et al., 2022) of cryptopnight. A multivariant package is a binary containing more than one hashing function. Concretely, the binaries `0d996462`, `d2141ff2`, `046dc081`, `a32a6f4b` and `fbdd1efa` contain between 2 and 3 versions of hashing functions `cryptonight_hash`.

Third, our manual analysis of the binaries reveals 6 main sources for these differences. (1) **Versions:** the binaries do not depend on the same version of cryptonight, (2) **Function reordering:** The order in which the functions are declared inside the binary changes (3) **Innocuous expressions:** Expressions have been injected into the program code, but their execution does not affect the semantic of the original program (4) **Function renaming:** The name of the functions exported to JavaScript have been changed (5) **Data layout changes:** The data needed by the cryptominers has different location in the WebAssembly linear memory. (6) **Partial cryptonight:** Some binaries exclude cryptonight functions, i.e., the `cryptonight_create`, `cryptonight_destroy`, `cryptonight_hash` functions. It is interesting to note that changes in function order, function names, or data layout leads to different programs that have the same number of instructions and functions, as is the case for 9 of our malign binaries.

5.3. Methodology

Based on our dataset of WebAssembly binaries, we follow the following procedures to answer our research questions.

RQ1: Evasion effectiveness To answer RQ1, we execute the baseline evasion algorithm discussed in Section 4.3. We first pass a suspicious binary to wasm-mutate. The diversified version is passed to VirusTotal as a binary oracle. If at least one vendor still detects the diversified binary, we pass it to wasm-mutate again to stack a new random transformation. We repeat this process until VirusTotal does not label the binary as malware or reach a limit of 1000 transformations. The process is performed 10 times with 10 different random seeds for each binary.

RQ2: Oracle minimization Malicious actors have a limited budget to perform evasion before they get caught. The number of oracle calls is a proxy for such a budget, i.e., the lesser the number of oracle calls, the lesser effort spent. To answer RQ2, we aim at minimizing the number of oracle calls while keeping the same evasion effectiveness. In particular, we assess the capability of the MCMC evasion algorithm to minimize the number of calls to the malware oracle (Metric 1). For this, we execute the MCMC evasion algorithm (Algorithm 2) one time for each malware binary of our dataset. We use VirusTotal as an oracle and stop when we reach a limit of 1000 transformations. Since MCMC has a configuration parameter σ, we repeat the process with three σ values: 0.01, 0.3, and 1.1. The σ-value weights exploration and exploitation of transformations during the evasion process. For example, the first value is low, favoring exploration at the most, meaning that the MCMC algorithm will take any new transformation, whether it increases the fitness function value or not. On the contrary, the largest value 1.1 favors the exploitation and, meaning that during evasion, MCMC will only accept transformations with higher values from the oracle, i.e., more evaded detectors. We manually select the third value 0.3 as the balance between exploration and exploitation.

RQ3: Malware functionality A diversified binary that fully evades the detection, might not be practical due to behavioral or performance issues. RQ3 complements our first two research questions with a correctness and an efficiency evaluation. For every cryptojacking that can be executed, we reproduce all cryptominer components (described in Section 2.2) and replace the WebAssembly binaries with variants that fully evade VirusTotal. For each executable cryptojacking program, we generate 10 variants with the baseline evasion algorithm as well as 10 variants with the MCMC algorithm, with σ value 0.3 to balance the MCMC exploration-exploitation. Then, we replace the original cryptojacking by each of the 20 variants in order to determine that the behavior of the original cryptojacking program is preserved in the variants (Demetrio et al., 2021).

Our end-to-end pipelines provide data on the number of generated hashes in the webpage component as an HTML element. Besides, the number of successful or incorrect jobs are logged by the miner pool. This information can be used to measure both the correctness and efficiency of the generated WebAssembly variants. To collect data on the number of hashes per second, the webpage is accessed with Puppetter, while the miner pool logs are saved to measure the number of successful and failed jobs with their respective log time. By analyzing these two types of data, the overall correctness and effectiveness of a diversified WebAssembly binary can be determined.

To check correctness, we verify that the hashes generated by the variants are valid. We determine whether the hashes reach the third component of a cryptojacking (see Section 2.2), i.e., how many successful and failed jobs are reported by the miner pool. If a miner pool correctly accepts the hashes, then the cryptojacking variant generated by our evasion algorithms is considered correct.

To check for efficiency, we measure the frequency of `hashes` produced by the variants binaries. For each WebAssembly cryptojacking and its generated variants, we execute and measure the hashes produced per second, during 1000 s. For each malware variant, we check if the hash production frequency is still in the same order of magnitude as the original.

[1] Binaries that could be found in a live webpage at the moment of this writing are marked with ● http:// .

[2] Yara project `https://github.com/davbo/yara-rs/tree/master/sample-miners`

[3] All Wasm binaries of the MinerRay project could be found at `https://github.com/miner-ray/miner-ray.github.io/tree/master/Data/SampleWasmFiles`

[4] All Wasm binaries of the SEISMIC project could be found at `https://github.com/wenhao1006/SEISMIC`

[5] Deepminer project `https://github.com/deepwn/deepMiner`

RQ4: Individual transformation effectiveness As discussed in Section 4.1, our diversifier comprises 135 possible transformation operators. In this research question, we want to study which transformations perform better in evading the VirusTotal malware detection oracle. This investigation will help future researchers and detectors engineers to improve their detection methods and tackle subversive transformations.

We use a value of $\sigma = 1.1$ to tune the MCMC evasion algorithm. With this parameter, the MCMC evasion algorithm only keeps transformations that significantly contribute to improving the fitness of the variant. In other words, a transformation is applied if at least one more detector of VirusTotal is evaded. Then, we count the number of applied transformations, aggregated by its type. By measuring this, we obtain the most used one (resp. the least used), and, therefore, understand where malware researchers should focus for counter-evasion.

RQ5: Effectiveness against MINOS This research question assesses the effectiveness of our evasion technique with respect to the state-of-the-art WebAssembly malware detector, MINOS (Naseem et al., 2021). This detector takes a Wasm binary as input and creates a 100×100 grayscale image from its pure byte stream. Using a Convolutional Neural Network, the generated image is classified as benign or malware.

We replicate MINOS. However, the model of MINOS is not publicly available, so we train our own model for this experiment. We use our own dataset to train the model with 33 malign programs and 33 benign programs. The 33 malign programs for training MINOS are the same listed in Table 1, the 33 benign programs are collected from the original MINOS reproduction steps. Our reproduction of MINOS achieves the same results as the original paper, based on one-off validation. Our reproduction of MINOS is publicly available at https://github.com/ASSERT-KTH/ralph.

We use MINOS as an oracle and follow the same method proposed in RQ1, passing each one of the binaries in Table 1 to our diversifier and then to MINOS. For each binary, we do the process 10 times with 10 different seeds, generating a total of 330 variants with no more than 1000 stacked mutations.

5.4. Metrics

In this section, we define the notions of total and partial evasion used in this work to measure the impact of the evasion algorithms proposed in Section 4.3. Besides, we also define the `number of oracle calls` and `number of stacked transformations` metrics. In addition, we define the metrics for correct hashes generated by the malware variants and the hashes generation speed.

Definition 1. Total evasion: Given a malware WebAssembly binary, an evasion algorithm generates a variant that totally evades detection if *all* detectors that originally identify the binary as malware identify the variant as benign.

Definition 2. Partial evasion: Given a malware WebAssembly binary, an evasion algorithm generates a variant that partially evades detection if *at least one* detector that originally identifies the binary as malware identifies the variant as benign.

Metric 1. Number of oracle calls: The number of calls made to the malware oracle during the evasion process.

Metric 2. Number of stacked transformations: The total number of transformations applied on the initial malware binary during the evasion process.

Notice that Metric 1 is the number of times that lines 4 and 5 in Algorithms 1 and 2 are executed, respectively. The same could be applied to Metric 2, in lines 7 and 13 of Algorithms 1 and 2, respectively.

The main purpose of a WebAssmbly cryptominer is to calculate hashes. By measuring the number of calculated hashes per time unit, we can measure how performant the cryptojacking is. Therefore, the impact of the evasion process over the performance of the created binary can be measured, calculating the number of hashes per time unit:

Metric 3. Crypto hashes per second (h/s): Given a WebAssembly cryptojacking, the crypto hashes per second metric is the number of successfully generated hashes in one second.

Metric 4. Correct crypto hashes: Given a WebAssembly cryptojacking, the number of correct crypto hashes is the number of hashes that the WebAssembly cryptojacking generates and that the miner pool accepts as valid.

6. Experimental results

In section, we answer our four research questions regarding the feasibility of WebAssembly diversification for malware evasion.

6.1. RQ1. Evasion effectiveness

We run our baseline evasion algorithm with a limit of 1000 iterations per binary. At each iteration, we query VirusTotal to check if the new binary evades the detection. This process is repeated with 10 random seeds per binary, resulting in a maximum of 10,000 queries per original binary. In total, we generate 98,714 variants for the original 33 suspicious binaries.

Table 2 shows the data to answer RQ1. The table contains as columns: the hash of the program, calculated as the sha256 hash, as its identifier, the number of initial VirusTotal detectors flagging the malware, the number of evaded antivirus vendors (cf. Definition 2) and the mean number of iterations needed to generate a variant that fully evades the detection (cf. Definition 1). The rows of the table are ordered with respect to the number of detectors for the original binary.

We observe that the baseline evasion algorithm successfully generates variants that totally evade detection for 30 out of 33 binaries. The mean value of iterations needed to generate a variant that evades all detectors ranges from 120 to 635 stacked transformations. For the 30 binaries that completely evade detection, we observe that the mean number of iterations to evade is correlated to the number of initial detectors. For example, the `a32a6f4b` binary, initially flagged by 18 detectors, requires around 635 iterations, while the `309c32c5`, with only one initial flag, needs 120 iterations. The mean number of iterations needed is always less than 1000 stacked transformations.

Figure 3 shows the evasion process with four different seeds for the binary `046dc081`. Each point in the x-axis represents 50 iterations, and the y-axis represents the number of VirusTotal detectors flagging the binary. Three out of 4 seeds manage to totally evade VirusTotal in less than 250 iterations. We have observed that there are better evasion techniques than pure random transformations. For example, the seed represented by the green line partially evades the oracle but shows no tendency to evade detection before 300 iterations. Besides, some transformations help some classifiers to detect the mutated binary. These phenomena are empirically exemplified in Fig. 3 in which the curves is not always monotonously decreasing, like the blue-colored curve. In this case, it goes from 3 VirusTotal detectors to 5 during the 50–100 iterations.

There are 3 binaries for which the baseline algorithm does not completely evade the detection. In these three cases, the algorithm misses 5 out 31, 6 out of 30 and 5 out 26 detectors. The explanation is the maximum number of iterations (1000) we use for our

Table 2
Baseline evasion algorithm for VirusTotal. The table contains as columns: the hash of the program, the number of initial VirusTotal detectors, the maximum number of evaded antivirus vendors and the mean number of iterations needed to generate a variant that fully evades detection. The rows of the table are sorted by the number of initial detectors, from left to right and top to bottom.

Hash	#D	Max. #evaded	Mean #trans.
47d29959	31	26 (83.8%)	N/A
9d30e7f0	30	24 (80.0%)	N/A
8ebf4e44	26	21 (80.7%)	N/A
dc11d82d	20	20 (100.0%)	355
0d996462	19	19 (100.0%)	401
a32a6f4b	18	18 (100.0%)	635
fbdd1efa	18	18 (100.0%)	310
d2141ff2	9	9 (100.0%)	461
aafff587	6	6 (100.0%)	484
046dc081	6	6 (100.0%)	404
643116ff	6	6 (100.0%)	144
15b86a25	4	4 (100.0%)	253
006b2fb6	4	4 (100.0%)	282
942be4f7	4	4 (100.0%)	200
7c36f462	4	4 (100.0%)	236
fb15929f	4	4 (100.0%)	297
24aae13a	4	4 (100.0%)	252
000415b2	3	3 (100.0%)	302
4cbdbbb1	3	3 (100.0%)	295
65debcbe	2	2 (100.0%)	131
59955b4c	2	2 (100.0%)	130
89a3645c	2	2 (100.0%)	431
a74a7cb8	2	2 (100.0%)	124
119c53eb	2	2 (100.0%)	104
089dd312	2	2 (100.0%)	153
c1be4071	2	2 (100.0%)	130
dceaf65b	2	2 (100.0%)	140
6b8c7899	2	2 (100.0%)	143
a27b45ef	2	2 (100.0%)	145
68ca7c0e	2	2 (100.0%)	137
f0b24409	2	2 (100.0%)	127
5bc53343	2	2 (100.0%)	118
e09c32c5	1	1 (100.0%)	120

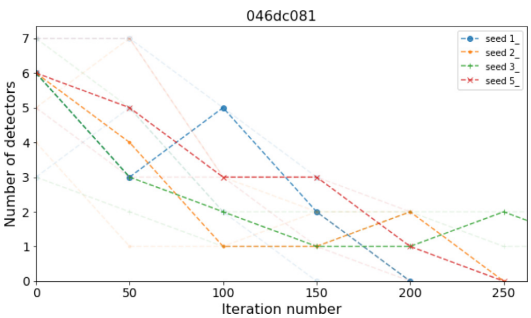


Fig. 3. The figure shows the evasion process with four seeds for binary `046dc081`. Each point in the *x*-axis represents 50 iterations, and the *y*-axis represents the number of VirusTotal detectors.

experiments. However, having more iterations seems not a realistic scenario. For example, if some transformations increment the binary size during the transformation, a considerably large binary might be impractical for bandwidth reasons.

On the other hand, there is a balance between generating variants and avoiding detection by defense mechanisms. For example, VirusTotal detects when it is being stressed with too many requests and too many queries can be detected and blocked, effectively ruining evasion. In our attack scenario, where VirusTotal is used as an external detector (see Section 3), we must be mindful of this limit. In contrast, when defense mechanisms such as MINOS are placed locally, we believe that the number of queries is not necessarily limited.

Wasm-mutate performs mutations based on the input binary. In the experiments for RQ1, the input binaries for the baseline algorithm comes from the application of a previous mutation. Yet, we have observed that some transformations can be applied in any order. This means that different sequences of transformations can produce the same binary variant. This often happens when two mutation targets inside the binary are different, such as two disjoint pieces of code. Therefore, a potential parallelization for the baseline algorithm is possible as soon as transformation sequences do not interfere with others.

Overall, our experiments prove that wasm-mutate is a powerful tool to perform malware evasion. By carefully selecting the order and type of transformations applied, it is possible to generate program variants that are both performant and effective at evading detection. This same idea is explored in the next section.

Answer to RQ1: The baseline evasion algorithm with wasm-mutate clearly decrease the detection rate by VirusTotal antivirus vendors for cryptojacking malware. We achieve total evasion of WebAssembly cryptojacking malware in 30/33 (90%) of our malware dataset.

6.2. RQ2. Oracle minimization

With RQ2, we analyze the effect of the MCMC evasion algorithm in minimizing the number of calls to the malware oracle (Metric 1). To answer RQ2, we execute the MCMC evasion algorithm discussed in Algorithm 2.

In Table 3 we can observe the impact of the MCMC evasion algorithm on our reference dataset. The first two columns of the table are the original program's hash and the number of initial VirusTotal detectors flagging the malware. The remaining columns are divided into two categories, maximum detectors evaded (Definition 2) and the number of oracle calls if total (Definition 1). Each one of the two categories contains the result for both evasion algorithms, first the baseline algorithm (BL) followed by the three σ-values analyzed in the MCMC evasion algorithm. Notice that, for the baseline algorithm, the number of oracle calls is the same value as the number of transformations needed to evade by construction. We highlight in bold text the values for which the baseline or the MCMC evasion algorithms are better than each other, the lower, the better.

We observe that the MCMC evasion algorithm successfully generates variants that totally evade the detection for 30 out of 33 binaries, it thus as good as the baseline algorithm. The improvement happens in the number of oracle calls. The oracle calls needed for the MCMC evasion algorithm are 92% of the needed on average for the baseline evasion algorithm.

For 21 of 30 binaries that evade detection entirely, we observe that the mean number of oracle calls needed is lower than those in the baseline evasion algorithm. For example, `f0b24409` needs 11 oracle calls with the MCMC evasion algorithm to fully evade VirusTotal, while for the baseline evasion algorithm, it needs 127 oracles calls. For those 21 binaries, it needs only 40% of the calls the baseline evasion algorithm needs.

The impact of the MCMC evasion algorithm is illustrated in Fig. 4. Each point in the *x*-axis represents 50 iterations, and the *y*-axis represents the number of VirusTotal detectors flagging binary 046dc081. 2 out of 3 σ-values manage to totally evade VirusTotal in less than 400 iterations. On the contrary, lower acceptance criteria $\sigma = 1.1$ (green line) partially evades the oracle, but does not fully evade within the maximum 1000 iterations limit of the experiment.

The σ value in the Algorithm 2 provides the acceptance criteria for new transformations in the MCMC evasion algorithm.

Table 3
MCMC evasion algorithm for VirusTotal. The first two columns of the table are: the identifier of the original program and the number of initial detectors. The remaining columns are divided into two categories, maximum detectors evaded if partial evasion and number of oracle calls if total evasion. Each one of the two categories contains the result of the evasion algorithms, first the baseline algorithm (BL) followed by the three σ-values analysed in the MCMC evasion algorithm. We highlight in bold text the values for which the baseline or the MCMC evasion algorithms are better from each other. Overall, the MCMC evasion algorithm needs less oracle calls than the baseline algorithm.

Hash	#D	Max. evaded				#Oracle calls			
		BL	MCMC			BL	MCMC		
			$\sigma = 0.1$	$\sigma = 0.3$	$\sigma = 1.1$		$\sigma = 0.01$	$\sigma = 0.3$	$\sigma = 1.1$
47d29959	31	**26**	19	12	10				
9d30e7f0	30	**24**	17	9	10				
8ebf4e44	26	**21**	13	5	4				
dc11d82d	20	**20**	**20**	14	15	**355**	446		
0d996462	19	**19**	**19**	14	4	**401**	697		
a32a6f4b	18	**18**	**18**	6	1	635	**625**		
fbdd1efa	18	**18**	**18**	3	3	**310**	726		
d2141ff2	9	**9**	**9**	**9**	5	**461**	781	783	
aafff587	6	**6**	2	**6**	**6**	484		**331**	413
046dc081	6	**6**	**6**	**6**	5	404	397	**159**	
643116ff	6	**6**	**6**	**6**	1	**144**	436	631	
15b86a25	4	**4**	**4**	**4**	**4**	253	208	214	**131**
006b2fb6	4	**4**	**4**	**4**	0	**282**	380	709	
942be4f7	4	**4**	**4**	**4**	**4**	**200**	**200**	**200**	219
7c36f462	4	**4**	**4**	2	0	236	**221**		
fb15929f	4	**4**	2	**4**	2	**297**		475	
24aae13a	4	**4**	**4**	**4**	0	**252**	401	446	
000415b2	3	**3**	**3**	**3**	2	302	376	**34**	
4cbdbbb1	3	**3**	**3**	**3**	**3**	295	204	**72**	685
65debcbe	2	**2**	**2**	**2**	**2**	131	**33**	**33**	**33**
59955b4c	2	**2**	**2**	**2**	**2**	130	**33**	**33**	**33**
89a3645c	2	**2**	**2**	**2**	**2**	431	319	197	**107**
a74a7cb8	2	**2**	**2**	**2**	**2**	124	**33**	**33**	**33**
119c53eb	2	**2**	**2**	**2**	**2**	104	45	480	**18**
089dd312	2	**2**	**2**	**2**	**2**	153	166	167	**123**
c1be4071	2	**2**	**2**	**2**	**2**	130	**33**	**33**	**33**
dceaf65b	2	**2**	**2**	**2**	**2**	140	166	166	**132**
6b8c7899	2	**2**	**2**	**2**	**2**	143	**33**	**33**	**33**
a27b45ef	2	**2**	**2**	**2**	**2**	145	**33**	**33**	**33**
68ca7c0e	2	**2**	**2**	**2**	**2**	137	**33**	167	595
f0b24409	2	**2**	**2**	**2**	**2**	127	**11**	33	**11**
5bc53343	2	**2**	**2**	**2**	**2**	118	**33**	**33**	**33**
e09c32c5	1	**1**	**1**	**1**	0	**120**	488	921	

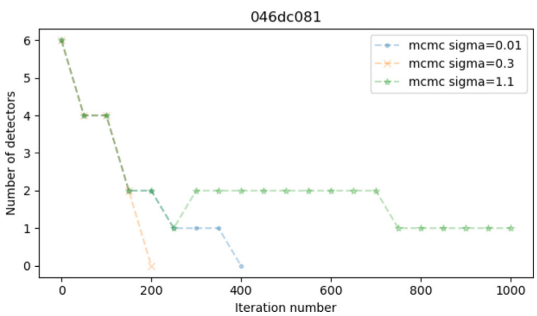


Fig. 4. The figure shows the MCMC evasion process for the binary 046dc081. Each point in the *x*-axis represents 50 iterations, and the *y*-axis represents the number of VirusTotal detectors. The figure shows less chaotic progress for oracle evasion.

We run the MCMC evasion algorithm with three values to better understand the extent to which the transformation space is explored to find a binary that evades successfully. We have observed that for a large value of σ, meaning low acceptance, 16 binaries cannot be mutated to evade VirusTotal entirely. The main reason is that, in this case, the MCMC evasion algorithm discards transformations that increase the number of oracle calls. Therefore, the algorithm gets stuck in local minima and never finds a binary that entirely evades. On the contrary, a small value of σ accepts more new transformations even if more detectors flag the binary.

According to our experiments, $\sigma = 0.3$ offer a good acceptance trade-off. However, the best value of σ actually depends on the binary to mutate. Therefore, we cannot conclude the best value of σ for the whole dataset. This is a consequence of the particularities of each one of the original binaries and its detectors. For example, we have observed that for the bottom part of Table 3 the highest value of σ works better overall. The main reason is the low number of original detectors. In those cases, the exploration space for transformations is smaller. Thus, for the MCMC evasion algorithm, the chances of a local minimum for the fitness function to be global are larger. Therefore, the MCMC evasion algorithm with low acceptance criteria can find the binary that fully evades in fewer iterations. On the contrary, if the number of initial detectors is more significant, the exploration space is too big to explore in 1000 max iterations. The reason is that the MCMC evasion algorithm applies one transformation per time. While we provide fine-grained analysis in our work, more than one transformation per iteration could be applied in the MCMC evasion algorithm to solve this.

In all 3 cases for which neither the baseline nor the MCMC evasion algorithms could find a binary that fully evades, the maximum evaded detectors are less for the MCMC evasion algorithm. The main reason is that the MCMC evasion algorithm might prevent transformations for which the number of detectors increases. As previously discussed, it is stuck in local minima, which means that it does not explore transformation paths for which a higher

Table 4
Malware correctness and efficiency. We execute each cryptojacking that can be reproduced, with the original malware, and with 10 baseline variants and 10 MCMC variants. The first left section indicates the identifier of the original binary and its original frequency of hash generation. The second section of the table shows correctness percentage and hashes per second for ten variants generated with our baseline algorithm. The third section of the table shows correctness percentage and hashes per second for ten variants generated with the MCMC algorithm. After each frequency of hashing measurement, we indicate the relative difference with the original frequency, in parentheses. A difference larger or equal to 1.0 indicates a variant that is faster or as efficient as the original; a difference lower than 1.0 is a variant slower than the original.

Hash	Original h/s	Baseline algorithm						MCMC algorithm					
0d996462	116.0	100%	25 (0.22)	100%	24 (0.21)	100%	26 (0.22)	100%	116 (**1.00**)	100%	70 (0.60)	100%	67 (0.58)
		100%	116 (**1.00**)	100%	110 (0.95)	100%	30 (0.26)	100%	110 (0.95)	100%	76 (0.66)	100%	60 (0.52)
		100%	55 (0.47)	100%	27 (0.23)	100%	23 (0.20)	100%	86 (0.74)	100%	60 (0.52)	100%	72 (0.62)
		100%	27 (0.23)					100%	76 (0.66)				
a32a6f4b	48.0	100%	25 (0.52)	100%	24 (0.50)	100%	24 (0.50)	100%	26 (0.54)	100%	45 (0.94)	100%	41 (0.85)
		100%	26 (0.54)	100%	25 (0.52)	100%	26 (0.54)	100%	46 (0.96)	100%	41 (0.85)	100%	45 (0.94)
		100%	26 (0.54)	100%	24 (0.50)	100%	25 (0.52)	100%	44 (0.92)	100%	42 (0.88)	100%	45 (0.94)
		100%	23 (0.48)					100%	45 (0.94)				
fbdd1efa	37.0	100%	25 (0.68)	100%	25 (0.68)	100%	25 (0.68)	100%	28 (0.76)	100%	47 (**1.27**)	100%	48 (**1.30**)
		100%	25 (0.68)	100%	26 (0.70)	100%	26 (0.70)	100%	47 (**1.27**)	100%	47 (**1.27**)	100%	53 (**1.43**)
		100%	25 (0.68)	100%	25 (0.68)	100%	25 (0.68)	100%	48 (**1.30**)	100%	48 (**1.30**)	100%	49 (**1.32**)
		100%	25 (0.68)					100%	47 (**1.27**)				
d2141ff2	113.0	100%	54 (0.48)	100%	55 (0.49)	100%	55 (0.49)	100%	107 (0.95)	100%	107 (0.95)	100%	107 (0.95)
		100%	57 (0.50)	100%	56 (0.50)	100%	56 (0.50)	100%	109 (0.96)	100%	106 (0.94)	100%	100 (0.88)
		100%	57 (0.50)	100%	53 (0.47)	100%	53 (0.47)	100%	101 (0.89)	100%	100 (0.88)	100%	107 (0.95)
		100%	55 (0.49)					100%	107 (0.95)				
046dc081	118.0	100%	58 (0.49)	100%	60 (0.51)	100%	59 (0.50)	100%	118 (**1.00**)	100%	120 (**1.02**)	100%	119 (**1.01**)
		100%	60 (0.51)	100%	55 (0.47)	100%	62 (0.53)	100%	120 (**1.02**)	100%	116 (0.98)	100%	120 (**1.02**)
		100%	55 (0.47)	100%	50 (0.42)	100%	57 (0.48)	100%	119 (**1.01**)	100%	120 (**1.02**)	100%	119 (**1.01**)
		100%	55 (0.47)					100%	120 (**1.02**)				
006b2fb6	8.0	100%	7 (0.88)	100%	6 (0.75)	100%	4 (0.50)	100%	6 (0.75)	100%	6 (0.75)	100%	6 (0.75)
		100%	9 (**1.12**)	100%	6 (0.75)	100%	4 (0.50)	100%	6 (0.75)	100%	6 (0.75)	100%	6 (0.75)
		100%	4 (0.50)	100%	6 (0.75)	100%	4 (0.50)	100%	8 (**1.00**)	100%	9 (**1.12**)	100%	6 (0.75)
		100%	6 (0.75)					100%	6 (0.75)				

number of detectors could lead to better long-term results and, eventually, full evasion.

Answer to RQ2: The MCMC evasion algorithm needs fewer oracle calls than the baseline algorithm. In 21 cases out of 33, it needs only 40% of oracle calls compared to the baseline, providing more stealthiness to the malicious organization directing the evasion. The acceptance criterion σ of the MCMC evasion algorithm needs to be carefully crafted depending on the original binary.

6.3. RQ3. Malware functionality

To answer RQ3, we select the six binaries we can build and execute end-to-end, because we have access to the three components previously mentioned in Section 2.2. For those six binaries, we are able to replace the original WebAssembly binary with variants generated by our evasion algorithms.

We execute the original binary and the variants generated by the baseline and MCMC evasion algorithms. The essence of a cryptojacking is to generate hashes at high-speed. Consequently, we assess the correctness of the variants concerning two properties: validity of the hashes and frequency of hash generation. With Metric 4, we determine the validity of the generated hashes by checking if the backend miner pool accepts them. The frequency is measured as the amount of `hashes` produced by the variant binaries in one second (Metric 3).

Table 4 summarizes the key data for RQ3. Each row of the table corresponds to one binary that can be executed end-to-end, by reproducing the three components mentioned in Section 2.2. The first two values of each row are the original binary's identifier and its original frequency of hash generation. Then, for our baseline algorithm, each row has the data for 10 variants generated during a successful oracle evasion. For each one of the variants, we include the correctness percentage and the hashes calculated per second. The last group of columns of the table contains the correctness percentage and the hashes calculated per second for the 10 variants generated with the MCMC algorithm. After each frequency of hashing measurement, we indicate the relative difference with the original frequency in parentheses. A difference larger or equal to 1.0 means that the variant is faster or as efficient as the original. On the contrary, the variant is slower if the difference is lower than 1.0. The table contains the information for 120 variants.

We use the backend miner pools (see Section 2.2) of the six cryptojacking to determine the validity of the hashes computed by all the programs. All correctness assessments for the 120 variants indicate that the miner pools do not detect any invalid hash when executing the WebAssembly cryptominer variants. This means that the variants synthesized by the baseline and the MCMC algorithms to evade the malware oracle can still systematically generate valid hashes.

We have observed that 23 of 120 variants are more efficient than the original cryptojacking. For example, for the `fbdd1efa` binary, all its variants are from 1.27 to 1.43 faster than the original hash generation frequency. This phenomenon occurs because wasm-mutate can perform transformations in the executable code, which work as optimizations. Our experiments have revealed two sources of faster WebAssembly variants: loop unrolling transformations and code replacements that lead to smaller binaries. We have also found that debloating transformations, which remove unneeded structures and dead code, results in more hashes being produced by the cryptominer in the first few seconds of mining, likely because of faster compilation.

In summary, all this is evidence that focused optimization is a good primitive for evasion. On the contrary, 97 out of 120 variants underperform compared to the original binary. The worst case is the binary `0d996462`. Its slowest variant has 0.20 of the original generation frequency. The main reason is that wasm-mutate also introduces non-optimal transformations regarding performance.

The variants generated by the baseline evasion algorithm tend to be slower than the MCMC evasion algorithm. The MCMC evasion algorithm triggers fewer oracle calls and can generate variants faster than the ones generated by the evasion algorithm. This phenomenon is a direct consequence of the MCMC evasion algo-

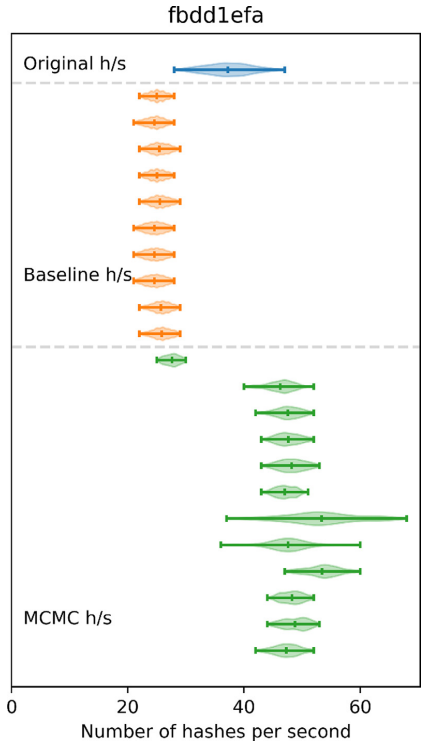


Fig. 5. Distribution of the number of hashes per second for the variants generated for the original `fbdd1efa` cryptojacking. In the figure we include the original binary (in blue), ten variants generated by the baseline algorithm (in orange), and ten variants generated by the MCMC evasion algorithm (in green). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

rithm implementation, which has a selective strategy when applying transformations on a binary (see Algorithm 2). In summary, the MCMC evasion algorithm produces variants that fully evade the VirusTotal oracle with lower performance overhead during execution. The worst performant variant is only 1.93 times slower for the MCMC evasion algorithm.

In Fig. 5, we plot the distribution of the hashes per second for variants generated for the `fbdd1efa` cryptojacking. In the figure, we include the original binary (in blue), the ten variants generated by the baseline algorithm (in orange), and the ten variants generated by the MCMC evasion algorithm (in green). Each violin plot corresponds to the hashes per second. We observe a normal distribution around the exact number of hashes per second. In this case, we have observed that the MCMC evasion algorithm provides a hash-per-second ratio better than the original. This phenomenon can be observed in the lines inside the violin plots, the green line is shifted to the right compared to the lines of the blue violin plot.

Answer to RQ3: Our algorithms synthesize WebAssembly cryptojacking variants that fully evade our malware oracle and that provide the same functionality as the original. The execution of evading malware systematically produces valid hashes, and the variations in performance are imperceptible. For 19% of the generated variants, we observe better performance, and in the worst case, the generated variant underperforms by five times the original binary.

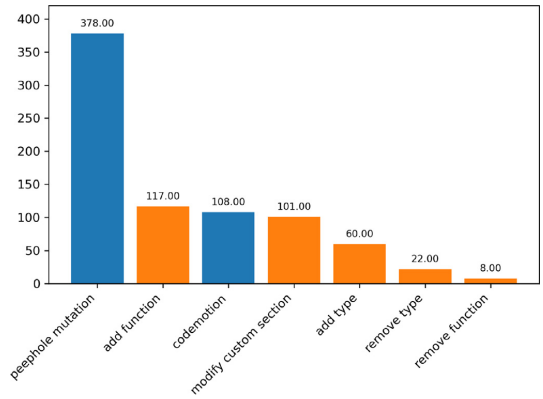


Fig. 6. Distribution of applied transformation for the MCMC evasion algorithm with $\sigma = 1.1$ (low acceptance). The *x*-axis displays the names of the transformations. The *y*-axis indicates the number of transformations found in the generated variants. We can observe which transformations perform better in order to provide total evasion.

6.4. RQ4. Individual transformation effectiveness

With RQ4, we investigate what individual transformations are the most appropriate to generate evading malware variants. Both attackers and defenders can leverage this information as follows. On the one hand, attackers know which transformation operators they can discard to speed-up the search for evading malware and minimize calls to the oracle. On the other hand, defenders can focus on the most effective transformations to evade antiviruses. For example, one way to defend against evasion is to use transformation as a preprocessing stage prior to detection. This can help to ensure that detection is more robust to potential evasion vectors.

In Fig. 6 we plot the distribution of transformations applied by the MCMC evasion algorithm with $\sigma = 1.1$ when it generates variants that fully evade the VirusTotal oracle. On the *x*-axis, we provide the name of the transformation, as wasm-mutate implements them. The *y*-axis is the absolute number of transformations found among the generated variants. The transformations are sorted in decreasing order of usage in the variants. We use two colors for transformations: transformations that affect the execution of the binary (in blue color) and transformations that do not (in orange color). For example, adding a new type definition does not affect the execution of the binary, while a peephole transformation does.

First, we concentrate on non-behavioral transformations in orange. The most effective transformation is to add a random function, which is present 117 times in the evading binaries. The next most present structural transformation is the `modify custom section` transformation. This highlights the sensitivity of malware detectors to the custom section. We note that custom section modification at the bytecode level provides advantages against source code obfuscation because we are sure that the compiler does not add metadata information that would help malware detectors. In other words, metadata information injected by compilers into WebAssembly binaries (Hilbig et al., 2021), could be removed from being part of possible detection.

Transformations `remove function` and `remove type` also affect malware detection. This novel observation indicates that VirusTotal is looking for code common in malware yet dead code. For example, malware detectors probably check for code that does not affect the original functionality of the binary. Thus, if this information is removed, the detector misses the malware. In other terms, by removing code, the detection surface is reduced.

```
...
local.get 0




i32.const 4
i32.add
...
```

Listing 4. Original piece of code for the 089a3645c WebAssembly binary.

```
...
local.get 0
i64.const -461681990785514485
drop
i32.const 0
i32.shr_u
i32.const 4
i32.add
...
```

Listing 5. Peephole mutations of Listing 4. Only with this mutation VirusTotal passed from 2 initial detectors to 0 in our experiments.

The most significant transformation to generate evading malware consists of peephole transformations (the largest bar of the figure at the left-hand side of the figure). The peephole transformations operate at the bytecode instruction level. For example, in Listings 4 and 5 we show a piece of the binary 089a3645c, and one peephole transformation applied, respectively. The transformation in the listings corresponds to a variant generated with the MCMC evasion algorithm. Only this transformation is required to fully evade for the 089a3645c binary.

Our results show that malware detectors should prioritize the detection of peephole transformations in WebAssembly, to increase the likelihood of detecting cryptojacking. For example, the transformation of Listing 5 can be reversed with static analysis.

When we answer our four research questions, we generate WebAssembly cryptominer variants by adding one transformation at a time (See Section 4.3). This method allows us to answer our research questions at a fine-grained level. For instance, the answer to RQ4 could only be possible if the transformations are analyzed one by one in the evasion process. Now, our method can be easily tuned to one-shot evasion: the algorithms could apply multiple transformations simultaneously to produce evading malware in one iteration. Consequently, the evasion process proposed in this work could be faster and more practical for a potential attacker. On the other hand, our algorithms stop as soon as one binary is diversified enough to provide total evasion. Since the overhead introduced by wasm-mutate is imperceptible, the transformation process can generate remarkably more binaries. Our approaches could escalate to infinite cryptojacking variants.

Answer to RQ4: Our experiments reveal that peephole transformations are the most effective for WebAssembly cryptojacking malware evasion. We also show that transformations on non-executable parts of WebAssembly binaries can contribute to evasion. These novel observations are crucial for cryptojacking malware detector vendors to prioritize their work on improving malware detection.

6.5. RQ5. Effectiveness against MINOS

To evaluate the effectiveness of MINOS at detecting diversified malware, we reuse the protocol of RQ1. We repeatedly stack random mutations to the original malware binary until MINOS is fully evaded or the maximum number of iterations is reached. We repeat this process 10 times for each binary. The results of our experiment are presented in Table 5. The table provides the identifier of the program and the mean number of iterations required to synthesize a variant that fully evades MINOS.

Table 5
The table provides the identifier of the program as its sha256 hash value and the mean number of iterations required to totally evade MINOS. Each binary was mutated with the baseline algorithm 10 times.

Hash	Mean #trans.	Hash	Mean #trans.
24aae13a	980.0	000415b2	960.0
59955b4c	38.0	119c53eb	1.0
fb15929f	1.0	5bc53343	33.0
47d29959	100.0	dc11d82d	115.0
a27b45ef	33.0	006b2fb6	1.0
942be4f7	29.0	7c36f462	85.0
0d996462	24.0	15b86a25	1.0
8ebf4e44	92.0	a74a7cb8	38.0
fbdd1efa	1.0	089dd312	68.0
65debcbe	38.0	aafff587	1.0
046dc081	33.0	6b8c7899	38.0
a32a6f4b	1.0	d2141ff2	81.0
68ca7c0e	38.0	dceaf65b	66.0
9d30e7f0	419.0	4cbdbbb1	1.0
643116ff	47.0	c1be4071	38.0
e09c32c5	15.0	f0b24409	33.0
89a3645c	108.0		

Our technique completely evades MINOS in all cases. In 2 cases out of 33, wasm-mutate needs more than 900 iterations to evade MINOS. The main reason is the application of symmetric mutations. For example, in some cases, wasm-mutate performs mutations that copy parts of the binary to another program location. Thus, when the binary is turned into a grayscale image, the embedding of the image is preserved, i.e., the code has changed, but the shape of the image has not. The contrary happens when non-symetric mutations are applied. For example, removing functions also removes embeddings of the grayscale image used by MINOS.

Remarkably, this experiment shows that WebAssembly diversification requires fewer iterations to evade MINOS than VirusTotal, meaning that it is easier to evade MINOS. The minimum number of iterations needed overall for evading VirusTotal are 118 for the baseline algorithm Table 2 and 11 for the MCMC algorithm Table 3, while for MINOS, wasm-mutate totally evades detection for 8 out of 33 binaries in one single iteration. This shows that the MINOS model is fragile wrt binary diversification. According to those results, VirusTotal can be considered better than MINOS wrt to cryptojacking detection.

To further enhance the detection capabilities of MINOS, we believe in binary canonicalization (Bruschi et al., 2007). By creating a canonical representation of the malware variant before training and inference, one would help the classifier to better generalize. This is feasible as it is a preprocessing step in the pipeline. We believe this is an interesting direction for future work.

Answer to RQ5: Our approach fully evades detection by the WebAssembly antivirus MINOS. In our study, we achieve evasion for all cryptojacking binaries in our dataset. wasm-mutate needs fewer iterations to totally evade MINOS compared to VirusTotal, validating VirusTotal as a baseline.

7. Discussion

In this section, we discuss the key challenges we faced, in order to help future research projects on similar topics.

Dataset size The dataset is smaller than other similar works for malware evasion. However, the related work does not consider WebAssembly – e.g. Ling et al. (2023) focus on Windows. For example,

while Tekiner and colleagues consider cryptojacking (Tekiner et al., 2021), we entirely focus only on WebAssembly cryptojacking malware. In this context, to the best of our knowledge, wasmbench is the most complete dataset of WebAssembly binaries. We analyze this dataset through the lens of VirusTotal and systematically extract all the cryptojacking malware it includes. Despite novelty, we acknowledge that the limited size of our malware dataset poses a challenge in terms of the generalizability of our results. Some types of malware might be absent from our dataset. To address this issue, one solution for future work would consist in expanding the dataset by using the inherent diversity found within the popular Cryptonight library. One could utilize the release history of their GitHub repository to compile and mine distinct, yet semantically equivalent malware instances. This approach would entail the exploration of a broader range of variations of cryptojacking malware Yet, this is considered as a new research paper per se as the process of mining and compiling code from a repository's release history is both time-consuming and computationally demanding. This is due to the need to analyze, filter, and compare a vast number of code commits and releases, as well as to validate their semantic equivalence. This process is even more complex in the case of malware reproduction due to the complex architecture in which this type of software operates (cf. Fig. 1.)

VirusTotal observations The final labelling of binaries for VirusTotal vendors is not definitive (Zhu et al., 2020). For example, a VirusTotal vendor could label a binary as benign and change it later to malign after several weeks. This phenomenon creates a time window in which slightly changed binaries (fewer iterations in our case) sometives evade the detection of numerous vendors. Also, we have observed that when our evasion algorithms manage to evade, some VirusTotal vendors result in timeout in several cases. This suggests that the evasion effectiveness is also due to performance constraints on the VirusTotal side.

Lack of abstraction A WebAssembly cryptojacking can only exist with its web complements. As we previously discussed, a browser cryptojacking needs to send the calculated hashes to a cryptocurrency service. This network communication is outside the WebAssembly accesses and needs to be delegated to a JavaScript code. We have observed that, the imports and the memory data of the WebAssembly binaries have a high variability in our dataset. The imported functions from JavaScript change from binary to binary. Their data segments also differ in content and length. This suggests that the whole JavaScript-WebAssembly polyglot package is the right direction for cryptojacking detection.

Mitigation As we noted in our response to RQ4 and RQ5, we believe that code canonicalization and is a promising mitigation technique if they are applied directly to WebAssembly. One way to do this would be to modify a diversifier such as wasm-mutate into a binary optimizer completely based on e-graph. This would provide canonicalization through a compact representation of WebAssembly code. In turn, malware variants with the same ancestor would be more seen as the “same” program, from its canonical representation.

8. Conclusion

We have demonstrated the potential for WebAssembly cryptojacking malware to be diversified and evade detection by leading malware detectors, such as VirusTotal and MINOS. Our generated variants are functional, performant, and do not trigger malware detection. Our evaluation of the technique against 60 state-of-the-art antiviruses through the meta-tool VirusTotal highlights the superiority of meta-antiviruses over single tailored defenses, even when the latter are specifically designed for WebAssembly cryptojacking binaries such as MINOS. By studying effective code transformations for evading cryptojacking detection, our work provides valuable insights and guidance for researchers in the field to better mitigate evasion.

As future work, we will improve the evasion fitness functions by including malware program properties, w.r.t. both evasion and malware execution performance. Some argue that the future of malware detection lies in machine learning. In future work, we believe in using our diversification technique to provide data augmentation for better malware detection.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

References

Afianian, A., Niksefat, S., Sadeghiyan, B., Baptiste, D., 2019. Malware dynamic analysis evasion techniques: asurvey. ACM Comput. Surv. 52 (6). doi:10.1145/3365001.

Aghakhani, H., Gritti, F., Mecca, F., Lindorfer, M., Ortolani, S., Balzarotti, D., Vigna, G., Kruegel, C., 2020. When malware is packin' heat; limits of machine learning classifiers based on static analysis features. In: Proc. of NDSS.

Aslan, O.A., Samet, R., 2020. A comprehensive review on malware detection approaches. IEEE Access 8, 6249–6271. doi:10.1109/ACCESS.2019.2963724.

Bhansali, S., Aris, A., Acar, A., Oz, H., Uluagac, A.S., 2022. A first look at code obfuscation for WebAssembly. In: Proc. of Conf. on Security and Privacy in Wireless and Mobile Networks doi:10.1145/3507657.3528560.

Bian, W., Meng, W., Zhang, M., 2020. Minethrottle: defending against wasm in-browser cryptojacking. In: Proceedings of The Web Conference 2020. Association for Computing Machinery doi:10.1145/3366423.3380085.

Bostani, H., Moonsamy, V., 2021. Evadedroid: a practical evasion attack on machine learning for black-box android malware detection. CoRR abs/2110.03301https://arxiv.org/abs/2110.03301.

Botacin, M., Ceschin, F., de Geus, P., Grégio, A., 2020. We need to talk about antiviruses: challenges & pitfalls of AV evaluations. Comput. Secur. 95. doi:10.1016/j.cose.2020.101859.

Botacin, M., Domingues, F.D., Ceschin, F., Machnicki, R., Zanata Alves, M.A., de Geus, P.L., Grégio, A., 2022. Antiviruses under the microscope: a hands-on perspective. Comput. Secur. 112. doi:10.1016/j.cose.2021.102500.

Bruschi, D., Martignoni, L., Monga, M., 2007. Code normalization for self-mutating malware. IEEE Secur. Privacy 5 (2), 46–54. doi:10.1109/MSP.2007.31.

Bytecodealliance, 2021. wasm-mutate. https://github.com/bytecodealliance/wasm-tools/tree/main/crates/wasm-mutate.

Cabrera Arteaga, J., Laperdrix, P., Monperrus, M., Baudry, B., 2022. Multi-variant execution at the edge. In: Proceedings of the 9th ACM Workshop on Moving Target Defense. Association for Computing Machinery doi:10.1145/3560828.3564007.

Cabrera-Arteaga, J., Malivitsis, O.F., Pérez, O.L.V., Baudry, B., Monperrus, M., 2021. Crow: code diversification for WebAssembly. In: Proceedings of MadWEB doi:10.14722/madweb.2021.23xxx.

Castro, R.L., Schmitt, C., Dreo, G., 2019. Aimed: evolving malware with genetic programming to evade detection. In: 2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE). IEEE, pp. 240–247.

Chua, M., Balachandran, V., 2018. Effectiveness of android obfuscation on evading anti-malware. In: Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy. Association for Computing Machinery doi:10.1145/3176258.3176942.

Cohen, F.B., 1993. Operating system protection through program evolution. Comput. Secur. 12 (6), 565–584.

Dasgupta, P., Osman, Z., 2021. A Comparison of State-of-the-Art Techniques for Generating Adversarial Malware Binaries. arXiv e-printsarXiv:2111.11487.

Demetrio, L., Biggio, B., Lagorio, G., Roli, F., Armando, A., 2021. Functionality-preserving black-box optimization of adversarial windows malware. IEEE Trans. Inf. Forensics Secur. 16, 3469–3478.

Egele, M., Scholte, T., Kirda, E., Kruegel, C., 2008. A survey on automated dynamic malware-analysis techniques and tools. ACM Comput. Surv. 44 (2). doi:10.1145/2089125.2089126.

GoogleLLC, 2022. Virustotal enterprise. https://assets.virustotal.com/vt-360-outcomes.pdf.

Haas, A., Rossberg, A., Schuff, D.L., Titzer, B.L., Holman, M., Gohman, D., Wagner, L., Zakai, A., Bastien, J., 2017. Bringing the web up to speed with WebAssembly. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation.

Hastings, W.K., 1970. Monte carlo sampling methods using Markov chains and their applications. Biometrika 57 (1), 97–109. http://www.jstor.org/stable/2334940

Hilbig, A., Lehmann, D., Pradel, M., 2021. An empirical study of real-world WebAssembly binaries: security, languages, use cases. In: Proceedings of the Web Conference 2021.
Kalash, M., Rochan, M., Mohammed, N., Bruce, N.D.B., Wang, Y., Iqbal, F., 2018. Malware classification with deep convolutional neural networks. In: 2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS), pp. 1–5. doi:10.1109/NTMS.2018.8328749.
Kaspersky, 2022. The state of cryptojacking in the first three quarters of 2022. https://securelist.com/cryptojacking-report-2022/107898/.
Kelton, C., Balasubramanian, A., Raghavendra, R., Srivatsa, M., 2020. Browser-based deep behavioral detection of web cryptomining with coinspy. In: Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb) 2020, pp. 1–12.
Kharraz, A., Ma, Z., Murley, P., Lever, C., Mason, J., Miller, A., Borisov, N., Antonakakis, M., Bailey, M., 2019. Outguard: detecting in-browser covert cryptocurrency mining in the wild. In: The World Wide Web Conference. Association for Computing Machinery doi:10.1145/3308558.3313665.
Konoth, R.K., Vineti, E., Moonsamy, V., Lindorfer, M., Kruegel, C., Bos, H., Vigna, G., 2018. Minesweeper: an in-depth look into drive-by cryptocurrency mining and its defense doi:10.1145/3243734.3243858.
Lachtar, N., Ibdah, D., Khan, H., Bacha, A., 2023. Ransomshield: a visualization approach to defending mobile systems against ransomware. ACM Trans. Priv. Secur. 26 (3). doi:10.1145/3579822.
Le, V., Afshari, M., Su, Z., 2014. Compiler validation via equivalence modulo inputs. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. Association for Computing Machinery doi:10.1145/2594291.2594334.
Li, D., Li, Q., Ye, Y.F., Xu, S., 2021. Arms race in adversarial malware detection: asurvey. ACM Comput. Surv. 55 (1). doi:10.1145/3484491.
Ling, X., Wu, L., Zhang, J., Qu, Z., Deng, W., Chen, X., Qian, Y., Wu, C., Ji, S., Luo, T., et al., 2023. Adversarial attacks against windows pe malware detection: a survey of the state-of-the-art. Comput. Secur. 103134.
Liu, L., Wang, B., 2016. Malware classification using gray-scale images and ensemble learning. In: 2016 3rd International Conference on Systems and Informatics (ICSAI), pp. 1018–1022. doi:10.1109/ICSAI.2016.7811100.
Lu, G., Debray, S.K., 2013. Weaknesses in defenses against web-borne malware - (short paper). In: Rieck, K., Stewin, P., Seifert, J. (Eds.), Detection of Intrusions and Malware, and Vulnerability Assessment - 10th International Conference, DIMVA. Proceedings doi:10.1007/978-3-642-39235-1_8.
Lu, L., Yegneswaran, V., Porras, P., Lee, W., 2010. Blade: an attack-agnostic approach for preventing drive-by malware infections. In: Proceedings of the 17th ACM conference on Computer and communications security, pp. 440–450.
Lundquist, G.R., Mohan, V., Hamlen, K.W., 2016. Searching for software diversity: attaining artificial diversity through program synthesis. In: Proceedings of the 2016 New Security Paradigms Workshop, pp. 80–91.
Monero, 2022. Monero. https://www.getmonero.org/.
Moser, A., Kruegel, C., Kirda, E., 2007. Limits of static analysis for malware detection. In: Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007), pp. 421–430. doi:10.1109/ACSAC.2007.21.
Mozilla, 2019. Protections Against Fingerprinting and Cryptocurrency Mining Available in Firefox Nightly and Beta. https://blog.mozilla.org/futurereleases/2019/04/09/protections-against-fingerprinting-and-cryptocurrency-mining-available-in-firefox-nightly-and-beta/.
Mozilla, 2022. Using web workers. https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers.
Musch, M., Wressnegger, C., Johns, M., Rieck, K., 2019a. New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild. 10.1007/978-3-030-22038-9_2
Musch, M., Wressnegger, C., Johns, M., Rieck, K., 2019. Thieves in the browser: web-based cryptojacking in the wild. In: Proceedings of the 14th International Conference on Availability, Reliability and Security. Association for Computing Machinery doi:10.1145/3339252.3339261.
Naseem, F., Aris, A., Babun, L., Uluagac, S., Tekiner, E., 2021. MINOS: A Lightweight Real-Time Cryptojacking Detection System. Ndss doi:10.14722/NDSS.2021.24444.
Payer, M., 2014. Embracing the new threat: towards automatically self-diversifying malware. In: The Symposium on Security for Asia Network, pp. 1–5.
Peng, P., Yang, L., Song, L., Wang, G., 2019. Opening the blackbox of virustotal: analyzing online phishing scan engines. In: Proceedings of the Internet Measurement Conference. Association for Computing Machinery doi:10.1145/3355369.3355585.
Ren, X., Ho, M., Ming, J., Lei, Y., Li, L., 2021. Unleashing the hidden power of compiler optimization on binary code difference: an empirical study. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. Association for Computing Machinery doi:10.1145/3453483.3454035.
Rokicki, T., Maurice, C., Botvinnik, M., Oren, Y., 2022. Port contention goes portable: port contention side channels in web browsers. In: Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security. Association for Computing Machinery doi:10.1145/3488932.3517411.
Romano, A., Lehmann, D., Pradel, M., Wang, W., 2022. Wobfuscator: obfuscating javascript malware via opportunistic translation to webassembly. In: Proceedings of the 2022 IEEE Symposium on Security and Privacy (S&P 2022), pp. 1101–1116.
Romano, A., Zheng, Y., Wang, W., 2020. Minerray: semantics-aware analysis for ever-evolving cryptojacking detection. In: 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 1129–1140.
Schkufza, E., Sharma, R., Aiken, A., 2012. Stochastic superoptimization. ACM SIGPLAN Notices 48. doi:10.1145/2451116.2451150.
Tekiner, E., Acar, A., Uluagac, A.S., Kirda, E., Selcuk, A.A., 2021. In-browser cryptomining for good: an untold story. In: 2021 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS), pp. 20–29. doi:10.1109/DAPPS52256.2021.00008.
VirusTotal, 2020. VirusTotal - Home. https://www.virustotal.com/gui/home/search.
Wang, W., Ferrell, B., Xu, X., Hamlen, K.W., Hao, S., 2018. Seismic: secure in-lined script monitors for interrupting cryptojacks. In: Lopez, J., Zhou, J., Soriano, M. (Eds.), Computer Security. Springer International Publishing, Cham, pp. 122–142.
Wang, W., Sun, R., Dong, T., Li, S., Xue, M., Tyson, G., Zhu, H., 2021. Exposing weaknesses of malware detectors with explainability-guided evasion attacks. arXiv preprint arXiv:2111.10085.
Willsey, M., Nandi, C., Remy Wang, Y., Flatt, O., Tatlock, Z., Panchekha, P., 2020. EGG: fast and extensible equality saturation. arXiv e-printsarXiv:2004.03082.
Xia, M., Gong, L., Lyu, Y., Qi, Z., Liu, X., 2015. Effective real-time android application auditing. In: Proceedings of the 2015 IEEE Symposium on Security and Privacy. IEEE Computer Society.
XMRIG, 2016. Xmrig. https://github.com/xmrig/xmrig.
Zhu, S., Shi, J., Yang, L., Qin, B., Zhang, Z., Song, L., Wang, G., 2020. Measuring and modeling the label dynamics of online anti-malware engines. 29th USENIX Security Symposium (USENIX Security 20). USENIX Association. https://www.usenix.org/conference/usenixsecurity20/presentation/zhu

Javier Cabrera-Arteaga is a Ph.D. student at KTH Royal Institute of Technology. His research interests are in the fields of Automated Software Engineering, Automated Testing and Software Diversification.

Martin Monperrus is Professor of Software Technology at KTH Royal Institute of Technology. His research lies in the field of software engineering with a current focus on automatic program repair, AI on code and program hardening. He received a Ph.D. from the University of Rennes, and a Master's degree from Compiègne University of Technology.

Tim Toady is a programmer analyst living in Merise, Estonia. His research interests include taint splatter analysis, Easter eggs and the usage of fakes in computing.

Benoit Baudry is a Professor in Software Technology at the KTH Royal Institute of Technology. His research focuses on automated software engineering, software diversity and software testing. He favors exploring code execution over code on disk.

WASM-MUTATE: FAST AND EFFECTIVE BINARY DIVERSIFICATION FOR WEBASSEMBLY

Javier Cabrera-Arteaga, Nick Fitzgerald, Martin Monperrus, Benoit Baudry
Under revision

WASM-MUTATE: Fast and Effective Binary Diversification for WebAssembly

Javier Cabrera-Arteaga[a,*], Nick Fitzgerald,[b], Martin Monperrus[a] and Benoit Baudry[a]

[a]KTH Royal Institute of Technology, Stockholm, Sweden
[b]Fastly Inc., San Francisco, USA

ABSTRACT

WebAssembly has quickly ascended as a promise for web development, renowned for its efficiency and expanding utility beyond browser environments. Yet, the burgeoning ecosystem of WebAssembly compilers and tools has created a need for robust software diversification techniques. Traditional approaches, often tied to specific compilation pipelines, can be restrictive in their scope and application.

We introduce WASM-MUTATE, a compiler-agnostic WebAssembly diversification engine. It is engineered to fulfill key criteria: the rapid generation of semantically equivalent yet behaviorally diverse WebAssembly variants, universal compatibility with existing WebAssembly programs, and the capability to counter high-risk security threats. Utilizing an e-graph data structure, WASM-MUTATE pioneers its usage in the realm of software diversification. Our assessments reveal that WASM-MUTATE can efficiently generate tens of thousands of unique WebAssembly variants in a matter of minutes. Additionally, it can produce variants with distinct machine code instruction traces and memory profiles in milliseconds. Notably, WASM-MUTATE's generated variants are fortified against Spectre attacks. This research not only delivers a potent tool for WebAssembly diversification but also contributes valuable insights for the development of more secure and resilient WebAssembly applications.

1. Introduction

WebAssembly is the fourth official language of the web, complementing HTML, CSS and JavaScript as a fast, platform-independent binary format [21, 40]. Since its introduction in 2015, it has seen rapid adoption, with support from all major browsers, including Firefox, Safari and Chrome. WebAssembly has also been adopted outside of browsers, with world-leading execution platforms like Fastly using it as a foundational technology for their content delivery network [17]. In addition to major ones like LLVM, more and more compilers and tools can output WebAssembly binaries [23, 45, 26]. With this prevalence, it is of utmost importance to design software protection techniques for WebAssembly [27].

Software diversification is a well-known software protection technique [12, 4, 19], consisting of producing numerous variants of an original program, each retaining equivalent functionality. Software diversification in WebAssembly has many important application domains, such as optimization [5] and malware evasion [8]. It can also been used for fuzzing, a salient example of this was the discovery of a CVE in Fastly in 2021 [18], achieved through automated transformations to a WebAssembly binary.

To develop an effective WebAssembly diversification engine, several key requirements must be met. First, the engine should be language-agnostic, enabling diversification of any WebAssembly code, regardless or the source programming language and compiler toolchain. Second, it must have the capability to swiftly generate semantically equivalent variants of the original code. The speed at which this diversification occurs holds potential for real-time applications, including moving target defense [6]. The engine should also possess the ability to counter attackers by producing sufficiently distinct code variants. This paper present an original system, WASM-MUTATE, that addresses all these requirements.

WASM-MUTATE is a tool to automatically transforms a WebAssembly binary program into a variant binary program that preserves the original functionality. The core of the diversification engine relies on an e-graph data structure [48]. To the best of our knowledge, this work is the first to use an e-graph for software diversification in WebAssembly. An e-graph offers one essential property for diversification: every path through the e-graph represents a functionally equivalent variant of the input program [48, 37]. A random e-graph traversal can also be very efficient, supporting the generation of tens of thousands of equivalent variants from a single seed program in minutes [29] Consequently, the choice of e-graphs is the key to build a diversification tool that is both effective and fast. We have designed 135 rewriting rules in WASM-MUTATE, which can transform the e-graph from fine to coarse grained levels.

We assess the effectiveness of WASM-MUTATE with respect to its capacity at generating variants, which code is different from the original and which execution exhibit diverse instruction and memory traces. Our empirical evaluation reuses an existing corpus from the diversification literature [7]. We also measure the speed at which WASM-MUTATE is able to generate the first variant that exhibits a trace different from the original. Our security security assessment of WASM-MUTATE consists in evaluating the degree to which diversification can mitigate Spectre attacks. This assessment

*Corresponding authors
javierca@kth.se (J. Cabrera-Arteaga); toady@eecs.kth.se (N. Fitzgerald,); monperrus@kth.se (M. Monperrus); baudry@kth.se (B. Baudry)
ORCID(s): 0000-0001-9399-8647 (J. Cabrera-Arteaga); 0000-0002-0209-2805 (N. Fitzgerald,); 0000-0003-3505-3383 (M. Monperrus); 0000-0002-4015-4640 (B. Baudry)

is made with WebAssembly programs that have been previously identified as vulnerable to Spectre attacks [36].

Our results demonstrate that WASM-MUTATE can generate thousands of variants in minutes. These variants have unique machine code after compilation with cranelift (static diversity) and the variants exhibit different traces at runtime (dynamic diversity). Our experiments also provide evidence that the generated variants are hardened against Spectre attacks. To sum up, the contributions of this work are:

- The design and implementation of a WebAssembly diversification pipeline, based on semantic-preserving binary rewriting rules.
- Empirical evidence on the diversity of variants created by WASM-MUTATE, both in terms of static binaries and execution traces.
- Demonstration that WASM-MUTATE can protect WebAssembly binaries against timing side-channel attacks, specifically, Spectre.
- An open-source repository, where WASM-MUTATE is publicly available for future research `https://github.com/bytecodealliance/wasm-tools/tree/main/crates/wasm-mutate`.

This paper is structured as follows. In section 2, we introduce WebAssembly, the concepts of semantic equivalence and what we state as a rewriting rule. In section 3, we explain and detail the architecture and implementation of WASM-MUTATE. We formulate our research questions in section 4, answering them in section 5. We discuss open challenges related to our research in section 6, in order to help future research projects on similar topics. In section 7 we highlight works related to our research on software diversification. We finalize with our conclusions section 8.

2. Background

In this section, we define and formulate the foundation of this work: WebAssembly and its runtime structure, semantic equivalence modulo input, rewriting rules and e-graphs. Along with the paper, we use the terms, metrics and concepts defined here.

2.1. WebAssembly

WebAssembly (Wasm) is a binary instruction set initially meant for the web, and now also used in the backend. It was adopted as a standardized language by the W3C in 2017, building upon the work of Haas et al. [21]. One of Wasm's primary advantages is that it defines its own Instruction Set Architecture (ISA), which is both platform-independent. As a result, a Wasm binary can execute on virtually any platform, including web browsers and server-side environments. WebAssembly programs are compiled ahead-of-time from source languages such as C/C++, Rust, and Go, utilizing compilation pipelines like LLVM.

```rust
fn main() {
    let mut arr = [1, 2, 3, 4, 5];
    // Variable assignment
    let mut sum = 0;
    // Loop and memory access
    for i in 0..arr.len() {
        sum += arr[i];
    }
    // Use of external function
    println!("Sum of array elements: {}", sum);
}
```

Listing 1: Rust program containing function declaration, loop, conditional and memory access.

```
(module
  (@custom "producer" "llvm.." )
  (import "env" "println" (func $println (param i32)))
  (memory 1)
  (export "memory" (memory 0))
  (func $main
    (local $sum i32)
    (local $i i32)
    (local $arr_offset i32)
    ; Initialize sum to 0 ;
    i32.const 0
    local.set $sum
    ; Initialize arr_offset to point to start of the array
          in memory ;
    i32.const 0
    local.set $arr_offset
    ; Initialize the array in memory;
    i32.const 0
    i32.const 1
    i32.store
    ...
    i32.store
    ...
    loop
      local.get $i
      i32.const 5
      i32.lt_s
      if
        ; Load array[i] and add to sum ;
        local.get $arr_offset
        local.get $i
        ...
        ; Increment i ;
        local.get $i
        i32.const 1
        i32.add
        local.set $i
        br 0
      else
        ; End loop ;
        i32.const 0
      end
    end

    ; Call external function to print sum ;
    local.get $sum
    call $println
  )
  ; Start the main function ;
  (start $main)
  )
```

Listing 2: Simplified WebAssembly code for Rust code in Listing 1.

WebAssembly programs operate on a virtual stack that allows primitive data types. Additionally, a WebAssembly

program might include several custom sections. For example, binary producers such as compilers use custom sections to store metadata, such as the name of the compiler that generates the Wasm code. A WebAssembly program also declares memory sections and globals, which are used to store, manipulate and share data during program execution, e.g. to share data with the host engine of the WebAssembly binary.

WebAssembly is designed with isolation as a primary consideration. For instance, a WebAssembly binary cannot access the memory of other binaries or cannnot interact directly with browser's APIs, such as the DOM or the network. Instead, communication with these features is constrained to functions imported from the host engine, ensuring a secure and safe Wasm environment. Moreover, control flow in WebAssembly is managed through explicit labels and well-defined blocks, which means that jumps in the program can only occur inside blocks, unlike regular assembly code [22]. In Listing 1, we provide an example of a Rust program that contains a function declaration, a loop, a loop conditional, and a memory access. When the Rust code is compiled to WebAssembly, it produces the code shown in Listing 2. The stack operations are folded with parentheses. The module in the example contains the components described previously.

The WebAssembly runtime structure is described in the WebAssembly specification and it includes 10 key elements: the Store, Stack, Locals, Module Instances, Function Instances, Table Instances, Memory Instances, Global Instances, Export Instances, and Import Instances. These components interact during the execution of a WebAssembly program, collectively defining the state of a program during its runtime.

Two of these elements, the Stack and Memory instances, are particularly significant in maintaining the state of a WebAssembly program during its execution. The Stack holds both values and control frames, with control frames handling block instructions, loops, and function calls. Meanwhile, Memory Instances represent the linear memory of a WebAssembly program, consisting of a contiguous array of bytes. In this paper, we highlight the aforementioned two components to define, compare and validate the state of two Wasm programs during their execution.

2.2. Semantic Equivalence

Semantic equivalence refers to the notion that two programs or functions are considered equivalent if, for a given specified input domain, they produce the same output values or have the same observable behavior [30]. In other words, the semantics of the two programs are equivalent when the input-output relationship (w/ possibly some abstraction), even if the internal implementation details or the structure of the programs differ.

Let us illustrate this with an example. Assume two programs P and P' (Listing 3 and Listing 4 respectively) where P' is the result of modifying a code in the first instruction of its unique function. The program P' has two extra instructions right before returning from the function. The remaining components of the original binary are not modified.

```
func (;0;) (type 0) (param i32
      f32) (result i64)
    i64.const 1
```

Listing 3: Program P.

```
func (;0;) (type 0) (param i32
      f32) (result i64)
    i64.const 1
    i32.const 42
    i32.drop
```

Listing 4: Program P', transformation of program P.

The state of the program P when entering the function is its stack $[S]$, the program P' has the same state before executing the function. The input values of the function for both programs are L, their outputs are the top of the stack at the end of the execution.

Program P has the state $[[S : i32.const\ 1]]$ just before returning from the function execution. When we trace the states of the program P', we can construct the following sequence of states:

1. $[[S : i32.const\ 1]]$ the integer constant 1 is now on the top of the stack.
2. $[[S : i32.const\ 1, i32.const\ 42]]$ the integer constant 32 is the top of the stack.
3. $[[S : i32.const\ 1]]$ the top of the stack is dropped. The function execution stops.

Notice that, the stack state of program P' is the same as program P. Thus, we can say that these two programs are semantically equivalent. Even though the programs share semantic equivalence, they display differences during execution. Specifically, P' stresses more on the stack by adding and subsequently dropping more values. These subtle yet significant differences form the crux of the diversification approaches discussed in this study.

2.3. Rewriting rule

Our definition of a rewriting rule draws from the one proposed by Sasnauskas et al. [41], and integrates a predicate to specify the replacement condition. Concretely, a rewriting rule is defined as a tuple, denoted as `(LHS, RHS, Cond)`. Here, `LHS` refers to the code segment slated for replacement, `RHS` is the proposed replacement, and `Cond` stipulates the conditions under which the replacement is acceptable. Importantly, `LHS` and `RHS` are meant to be semantically equivalent, per the definition of previous section.

For example, the rewriting rule `(x, x i32.or x, {})` implies that the `LHS` 'x' is to be replaced by an idempotent bitwise `i32.or` operation with itself, absent any specific conditions. Notice that, for this specific rule, the commutative property shared by `LHS` and `RHS`, symbolized as `(LHS, RHS)=(RHS, LHS)`. Besides, the `Cond` element could be an arbitrary criterion. For instance, the condition for applying the aforementioned rewriting rule could be to ensure that the newly created binary file does not exceed a threshold binary size.

Based on our understanding, our research is the first to apply the concept of rewriting rules to WebAssembly. This

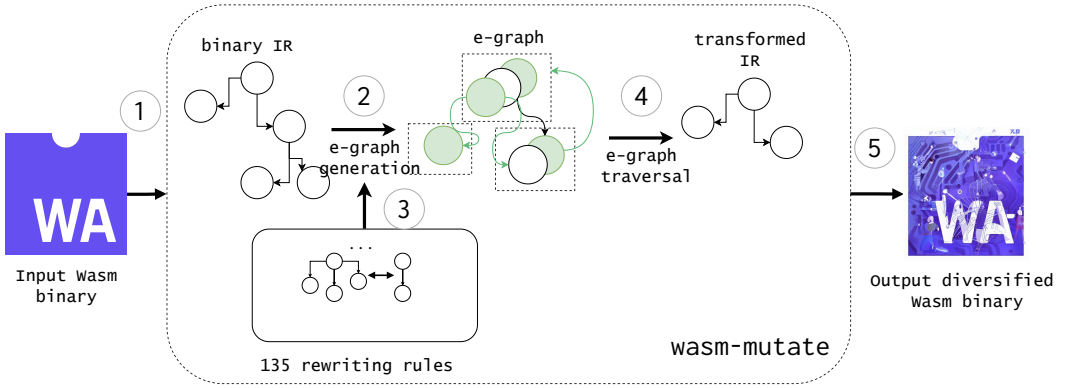


Figure 1: WASM-MUTATE workflow and high level architecture. It generates semantically equivalent variants from a given WebAssembly binary input. Its central approach involves synthesizing these variants by substituting parts of the original binary using rewriting rules, boosted by a diversification space traversals using e-graphs(refer to subsection 3.3)

will expand the potential use cases of wasm-mutate. Beyond its role as a diversification tool, it can also be used as a standard tool for conducting program transformations in WebAssembly.

3. Design of Wasm-Mutate

In this section we present WASM-MUTATE, a tool to diversify WebAssembly binaries and produce semantically equivalent variants.

3.1. Overview

The primary objective of WASM-MUTATE is to perform diversification, i.e., generate semantically equivalent variants from a given WebAssembly binary input. WASM-MUTATE's central approach involves synthesizing these variants by substituting parts of the original binary using rewrite rules. It leverages a comprehensive set of rewrite rules, boosted by a diversification space traversals using e-graphs(refer to subsection 3.3).

In Figure 1 we illustrate the workflow of WASM-MUTATE: it starts with a WebAssembly binary as input ①. It parses the original binary ②, turning the input program into appropriate abstractions, in particular WASM-MUTATE builds the control flow graph and data flow graph. Using the defined rewriting rules, WASM-MUTATE builds an e-graph ③ for the original program. An e-graph packages every possible equivalent code derivable from the given rewriting rules [48, 37]. Thus, at this stage, WASM-MUTATE exploits a key property of e-graphs: any path traversal through the e-graph results in a semantically equivalent code. Then, the diversification process starts, with parts of the original program being randomly replaced by traversal of the e-graph ④. The outcome of WASM-MUTATE is a semantically equivalent variant of the original binary ⑤. The tool guarantees semantically equivalent variants because each individual rewrite rule is semantic preserving.

3.2. WebAssembly Rewriting rules

In total, there are 135 possible rewriting rules implemented in WASM-MUTATE, those rules are grouped under several categories, called hereafter meta-rules. For example, 125 rewriting rules are implemented as part of a peephole meta-rule. There are 7 meta-rules that we present next.

Add type: In WebAssembly, the type section wraps definitions of signatures for the binary functions. WASM-MUTATE implements two rewrite rules, one of which is illustrated in the following rewriting rule.

```
LHS (module
  (type (;0;) (func (param i32) (result i64)))

---

RHS (module
  (type (;0;) (func (param i32) (result i64)))
+ (type (;0;) (func (param i64) (result i32 i64)))
```

This transformation generates random function signatures with a random number of parameters and results count. This rewriting rule does not affect the runtime behavior of the variant. It also guarantees that the index of the already defined types is consistent after the addition of a new type. This is because Wasm programs cannot access or use a type definition during runtime, they are only used to validate the signature of a function during compilation and validation from the host engine. From the security perspective, this transformation prevents against static binary analysis. For example, to avoid malware detection based on signature set [8].

Add function: The function and code sections of a Wasm binary contain function declarations and the code body of the declared functions, respectively. WASM-MUTATE add new functions, through mutations in the two mentioned sections. To add a new function, WASM-MUTATE creates a random type signature. Then, the random

function body is created. The body of the function consists of returning the default value of the result type. The following example illustrates this rewriting rule.

```
LHS (module
   (type (;0;) (func (param i32 f32) (result i64)))
```

```
RHS (module
   (type (;0;) (func (param i32 f32) (result i64)))
+_______(func (;0;) (type 0) (param i32 f32) (result i64)
+__________i64.const 0)
```

WASM-MUTATE never adds a call instruction to this function. So in practice, the new function is never executed. Therefore, executing both, the original binary and the mutated one with the same input, lead to the same final state. This strategy follows the work of Cohen, advocating the insertion of harmless 'garbage' code into a program. These transformations do not impact the program's functionality; they increase its static complexity.

Remove dead code: WASM-MUTATE can randomly remove dead code. In particular WASM-MUTATE removes: *functions, types, custom sections, imports, tables, memories, globals, data segments and elements* that can be validated as dead code with guarantees. For instance, to delete a memory declaration, the binary code must not contain a memory access operation. Separate mutators are included within WASM-MUTATE for each of the aforementioned elements. For a more concrete example, the following listing illustrates the case of a function removal.

```
LHS (module (type (func)))
```

```
RHS - (module (import "" "" (func)))

Cond The removed function is not called, it is not exported, and it is
       not in the binary _table.
```

When removing a function, WASM-MUTATE ensures that the resulting binary remains valid and semantically identical to the original binary: it checks that the deleted function was neither called within the binary code nor exported in the binary external interface. As exemplified above, WASM-MUTATE might also eliminate a function import while removing the function.

Eliminating dead code serves a dual purpose: it minimizes the attack surface available to potential malicious actors [1] and strengthens the resilience of security protocols. For instance, it can obstruct signature-based identification [8]. With Narayan and colleagues having demonstrated the feasibility of Return-Oriented Programming (ROP) attacks [36], the removal of dead code is able to stop jumps to harmful behaviors within the binary. On the other hand, the act of removing dead code reduces the binary's size, improving its non-functional properties, in particular bandwidth constraints.

Edit custom sections: The custom section in WebAssembly is used to store metadata, such as the name of the compiler that produces the binary or the symbol information for debugging. Thus, this section does not affect the execution of the Wasm program. WASM-MUTATE includes one mutator to edit custom sections. This is exemplified in the following rewriting rule.

```
LHS (module
...
   - (@custom "CS42" "zzz...")
```

```
RHS (module
...
   + (@custom "CS42" "xxx...")
```

The *Edit Custom Section* transformation operates by randomly modifying either the content or the name of the custom section. As illustrated by Cabrera-Arteaga et al. [8], such a rewriting strategy also acts as a potent deterrent against compiler identification techniques. Furthermore, it can also be employed in an innovative manner to emulate the characteristics of a different compiler, *masquerading* as another compilation source. This strategy ultimately aids in shrinking the identification and fingerprinting surface accessible to potential adversaries, hence enhancing overall system security, or to make it a moving target.

If swapping: In WebAssembly, an if-construction consists of a consequence and an alternative. The branching condition is executed right before the `if` instruction; if the value at the top of the stack is greater than `0`, then the consequence-code is executed, otherwise the alternative-code is run. The *if swapping* rewriting swaps the consequence and alternative codes of an if-construction.

To swap an if-construction in WebAssembly, WASM-MUTATE inserts a negation of the value at the top of the stack right before the `if` instruction. In the following rewriting rule we show how WASM-MUTATE performs this rewriting.

```
LHS (module
   (func ...) (
condition C
      (if A else B end)
   )
)
```

```
RHS (module
   (func ...) (
condition C
i32.eqz
      (if B else A end)
   )
)
```

The consequence and alternative codes are annotated with the letters `A` and `B`, respectively. The condition of the if-construction is denoted as `C`. The negation of the condition is achieved by adding the `i32.eqz` instruction in the RHS part of the rewriting rule. The `i32.eqz` instruction compares the top value of the stack with zero, pushing the value `1` if the comparison is true. Some if-constructions may not have either a consequence or an alternative code. In such cases, WASM-MUTATE replaces the missing code block with a single `nop`

instruction. In the context of ROP [36], this transformation can protect a victim binary to be exploited.

Loop Unrolling: Loop unrolling is a technique employed to enhance the performance of programs by reducing loop control overhead [14]. WASM-MUTATE incorporates a loop unrolling transformation and utilizes the Abstract Syntax Tree (AST) of the original Wasm binary to identify loop constructions.

When WASM-MUTATE selects a loop for unrolling, its instructions are divided by first-order breaks, which are jumps to the loop's start. This separation ensures that branching instructions controlling the loop body do not require label index adjustments during unrolling. The same holds true for instructions continuing to the next loop iteration. As the loop unrolling process unfolds, a new Wasm block is created to encompass both the duplicated loop body and the original loop. Within this newly established block, the previously separated groups of instructions are copied. These replicated groups of instructions mirror the original ones, except for branching instructions jumping outside the loop body, which need their jumping indices increased by one. This modification is required due to the introduction of a new `block ... end` scope around the loop body, which affects the scope levels of the branching instructions.

In the following text we illustrate the rewriting rule for a function that contains a loop.

```
LHS (module
    (func ...) (
        (loop   A br_if 0   B end)
    )
)
```

```
RHS (module
    (func ...) (
        (block
            (block A'   br_if 0   B'   br 1   end)
            (loop   A'   br_if 0   B'   end)
        end)
    )
)
```

The loop in the LHS part features a single first-order break, indicating that its execution will cause the program to continue iterating through the loop. The loop body concludes right before the `end` instruction, which highlights the point at which the original loop breaks and resumes program execution. Upon selecting the loop for unrolling, its instructions are divided into two groups, labeled `A` and `B`. As illustrated in the RHS part, the unrolling process entails creating two new Wasm blocks. The outer block encompasses both the original loop structure and the duplicated loop body, while the inner blocks, denoted as `A'` and `B'`, represent modifications of the jump instructions in groups `A` and `B`, respectively. Notice that, any jump instructions within `A'` and `B'` that originally leaped outside the loop must have their jump indices incremented by one. This adjustment accounts for the new block scope introduced around the loop body during the unrolling process. Furthermore, an unconditional branch is placed at the end of the unrolled loop iteration's body. This ensures that if the loop body does not continue, the tool breaks out of the scope instead of proceeding to the non-unrolled loop.

Loop unrolling enhances resistance to static analysis while maintaining the original performance [38]. In particular, Crane et al. [13] have validated the effectiveness of adding and modifying jump instructions against Function-Reuse attacks. Our rewriting rule has the same advantages, it unrolls loops while 1) incorporating new jumps and 2) editing existing jumps, as it can be observed with the addition of the `br_if`, `end`, and `br` instructions.

Peephole: This transformation category is about rewriting instruction sequences within function bodies, signifying the most granular level of rewriting. We implement 125 rewriting rules for this group in WASM-MUTATE. We include rewriting rules that affects the memory of the binary. For example, we include rewriting rules that creates random assignments to newly created global variables. For these rules, we incorporate several conditions, denoted by `Cond`, to ensure successful replacement. These conditions can be utilized interchangeably and combined to constrain transformations (see subsection 3.3).

For instance, WASM-MUTATE is designed to guarantee that instructions marked for replacement are deterministic. We specifically exclude instructions that could potentially cause undefined behavior, such as function calls, from being mutated. For this rewriting type, WASM-MUTATE only alters stack and memory operations, leaving the control frame labels unaffected.

The peephole category rewriting rules are meticulously designed and manually verified. An instance of such streamlined transformation can is illustrated in subsection 2.3, `( x i32.or x, x, {})` implies that the `LHS` 'x' is to be replaced by an idempotent bitwise `i32.or` operation with itself, in the absence of any specific conditions. Therefore, this category continues to uphold the benefits previously discussed under the *Remove Dead Code* category.

3.3. E-graphs for WebAssembly

We build WASM-MUTATE on top of e-graphs. An e-graph is a graph data structure utilized for representing rewriting rules [9] and their chaining. In an e-graph, there are two types of nodes: e-nodes and e-classes. An e-node represents either an operator or an operand involved in the rewriting rule, while an e-class denotes the equivalence classes among e-nodes by grouping them, i.e., an e-class is a virtual node compound of a collection of e-nodes. Thus, e-classes contain at least one e-node. Edges within the graph establish operator-operand equivalence relations between e-nodes and e-classes.

In WASM-MUTATE, the e-graph is automatically built from a WebAssembly program by analyzing its expressions and operations through its data flow graph. Then, each unique expression, operator, and operand are transformed into e-nodes. Based on the input rewriting rules, the equivalent expressions are detected, grouping equivalent e-nodes into e-classes. During the detection of equivalent expres-

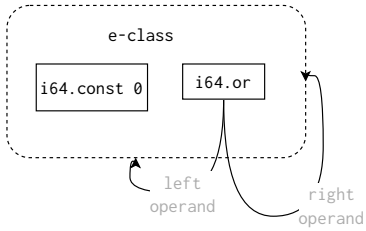


Figure 2: e-graph for idempotent bitwise-or rewriting rule. Solid lines represent operand-operator relations, and dashed lines represent equivalent class inclusion.

sions, new operators could be added to the graph as e-nodes. Finally, e-nodes within an e-class are connected with edges to represent their equivalence relationships.

For example, let us consider one program with a single instruction that returns an integer constant, `i64.const 0`. Let us also assume a single rewriting rule, `(x, x i64.or x, x instanceof i64)`. In this example, the program's control flow graph contains just one node, representing the unique instruction. The rewriting rule represents the equivalence for performing an `or` operation with two equal operands. Figure 2 displays the final e-graph data structure constructed out of this single program and rewriting rule. We start by adding the unique program instruction `i64.const 0` as an en e-node (depicted by the leftmost solid rectangle node in the figure). Next, we generate e-nodes from the rewriting rule (the rightmost solid rectangle) by introducing a new e-node, `i64.or`, and creating edges to the `x` e-node. Following this, we establish equivalence. The rewriting rule combines the two e-nodes into a single e-class (indicated by the dashed rectangle node in the figure). As a result, we update the edges to point to the `x` symbol e-class.

Willsey et al. illustrate that the extraction of code fragments from e-graphs can achieve a high level of flexibility, especially when the extraction process is recursively defined through a cost function applied to e-nodes and their operands. This approach guarantees the semantic equivalence of the extracted code [48]. For example, to obtain the smallest code from an e-graph, one could initiate the extraction process at an e-node and then choose the AST with the smallest size from among the operands of its associated e-class [35]. When the cost function is omitted from the extraction methodology, the following property emerges: *Any path traversed through the e-graph will result in a semantically equivalent code variant*. This concept is illustrated in Figure 2, where it is possible to construct an infinite sequence of "or" operations. In the current study, we leverage this inherent flexibility to generate mutated variants of an original program. The e-graph offers the option for random traversal, allowing for the random selection of an e-node within each e-class visited, thereby yielding an equivalent expression.

We propose and implement the following algorithm to randomly traverse an e-graph and generate semantically

Algorithm 1 e-graph traversal algorithm.

```
1:  procedure TRAVERSE(egraph, eclass, depth)
2:      if depth = 0 then
3:          return smallest_tree_from(egraph, eclass)
4:      else
5:          nodes ← egraph[eclass]
6:          node ← random_choice(nodes)
7:          expr ← (node, operands = [])
8:          for each child ∈ node.children do
9:              subexpr ← TRAVERSE(egraph, child, depth − 1)
10:             expr.operands ← expr.operands ∪ {subexpr}
11:         return expr
```

equivalent program variants, see 1. It receives an e-graph, an e-class node (initially the root's e-class), and the maximum depth of expression to extract. The depth parameter ensures that the algorithm is not stuck in an infinite recursion. We select a random e-node from the e-class (lines 5 and 6), and the process recursively continues with the children of the selected e-node (line 8) with a decreasing depth. As soon as the depth becomes zero, the algorithm returns the smallest expression out of the current e-class (line 3). The subexpressions are composed together (line 10) for each child, and then the entire expression is returned (line 11). To the best of our knowledge, WASM-MUTATE, is the first practical implementation of random e-graph traversal for WebAssembly.

Let's demonstrate how the proposed traversal algorithm can generate program variants with an example. We will illustrate Algorithm 1 using a maximum depth of 1. Listing 5 presents a hypothetical original Wasm binary to mutate. In this example, the developer has established two rewriting rules: `(x, x i32.or x, x instanceof i32)` and `(x, x i32.add 0, x instanceof i32)`. The first rewriting rule represents the equivalence of performing an `or` operation with two equal operands, while the second rule signifies the equivalence of adding 0 to any numeric value. By employing the code and the rewriting rules, we can construct the e-graph depicted in Figure 3. The figure demonstrates the operator-operand relationship using arrows between the corresponding nodes.

```
(module
   (type (;0;) (func (param i32 f32) (result i64)))
   (func (;0;) (type 0) (param i32 f32) (result i64)
      i64.const 1)
)
```

Listing 5: Wasm function.

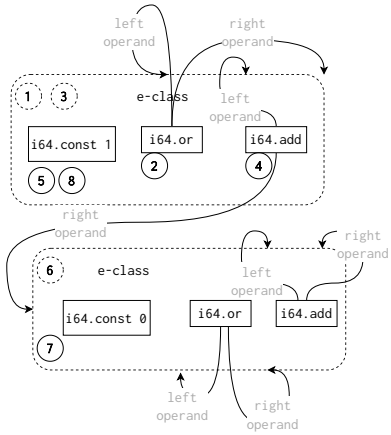


Figure 3: e-graph built starting in the first instruction of Listing 5.

```
(module
    (type (;0;) (func (param i32 f32) (result i64)))
    (func (;0;) (type 0) (param i32 f32) (result i64)
        (i64.or (
            (i64.add (
                i64.const 0
                i64.const 1
            ))
            i64.const 1
        ))
    )
```

Listing 6: Random peephole mutation using egraph traversal for Listing 5 over e-graph Figure 3. The textual format is folded for better understanding.

In Figure 3, we annotate the various steps of Algorithm 1 for the scenario described above. Algorithm 1 begins at the e-class containing the single instruction `i64.const 1` from Listing 5. It then selects an equivalent node in the e-class (2), in this case, the `i64.or` node, resulting in: `expr = i64.or l r`. The traversal proceeds with the left operand of the selected node (3), choosing the `i64.add` node within the e-class: `expr = i64.or (i64.add l r) r`. The left operand of the `i64.add` node is the original node (5): `expr = i64.or (i64.add i64.const 1 r) r`. The right operand of the `i64.add` node belongs to another e-class, where the node `i64.const 0` is selected (6)(7): `expr = i64.or (i64.add i64.const 1 i64.const 0) r`. In the final step (8), the right operand of the `i64.or` is selected, corresponding to the initial instruction e-node, returning: `expr = i64.or (i64.add i64.const 1 i64.const 0)i64.const 1` The traversal result applied to the original Wasm code can observed in Listing 6.

3.4. WASM-MUTATE in practice

In practice, WASM-MUTATE serves as a module within a broader process. This process starts from a WebAssembly binary as input and iterates over the variants generated by WASM-MUTATE in order to provide guarantees. In particular, it ensures that the output variant exhibits a different machine code per the JIT engine that executes it and unique execution traces when running. This process is explicitly laid out in Algorithm 2. One of the key elements in this algorithm is line 8, which activates WASM-MUTATE's diversification engine.

The algorithm starts by running the original WebAssembly program and recording its original execution traces, as denoted in line 5. These initial traces act as a reference for evaluating subsequent variants. An budget-based loop then initiates, as marked by lines 8 and 9, aiming to apply a series of code transformations. Upon the successful creation of a unique variant, line 11 triggers a JIT compilation within the WebAssembly engine. This step compiles the variant into machine code. The algorithm next assesses whether this machine code diverges from the original, thus confirming the actual diversity. If this condition is satisfied, the algorithm executes the variant to collect its low-level execution traces. The loop ends when a variant is found with new traces that are distinct from the original, as validated in line 15. The algorithm then returns the generated variant, wich guarantees that both diversified machine code and traces are different from the original.

Algorithm 2 WASM-MUTATE in practice.

```
1:  procedure DIVERSIFY(originalWasm, engine)
2:  Input:    ▷ A WebAssembly binary to diversify and a WebAssembly engine.
3:  Output:   ▷ A statically unique and behaviourally different WebAssembly variant.
4:
5:      originalTrace ← engine.execute(originalWasm)
6:      wasm ← originalWasm
7:      while true do
8:          variantWasm ← WASM-MUTATE(wasm)
9:          wasm ← variantWasm // we stack the transformation
10:         if variantWasm is unique then
11:             variantJIT ← engine.compile(variantWasm)
12:             if variantJIT is unique then
13:
14:                 trace ← engine.execute(variantJIT)
15:                 if trace ≠ originalTrace then
16:                     return variantWasm
```

3.5. Implementation

WASM-MUTATE is implemented in Rust, comprising approximately, 10 thousands lines of Rust code. We leverage the capabilities of the wasm-tools project of the bytecodealliance for parsing and transforming WebAssembly binary code. Specifically, we utilize the wasmparser and wasm-encoder modules for parsing and encoding Wasm binaries, respectively. The implementation of WASM-MUTATE is publicly availabel for future research and can be found at `https://github.com/bytecodealliance/wasm-tools/tree/main/crates/wasm-mutate`.

Source	Program	RQ	#F	# Ins.	Attack
CROW [7]	303	RQ1, RQ2	7-103	170-36023	N/A
Swivel [36]	btb_breakout	RQ3	16	743	Spectre branch target buffer (btb)
Swivel [36]	btb_leakage	RQ3	16	297	Spectre branch target buffer(btb)
Safeside [36, 20]	ret2spec	RQ3	2977	378894	Spectre Return Stack Buffer (rsb)
Safeside [36, 20]	pht	RQ3	2978	379058	Spectre Pattern History Table (pht)

Table 1
WebAssembly dataset used to evaluate WASM-MUTATE. Each row in the table corresponds to programs, with the columns providing: where the program is sourced from, the number of programs, research question addressed, function count, the total number of instructions found in the original WebAssembly program and the type of attack that the original program was subjected to.

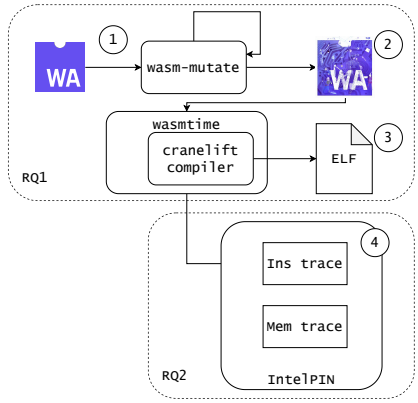


Figure 4: Protocol to answer RQ1 and RQ2

4. Evaluation

In this section, we outline our methodology for evaluating WASM-MUTATE. Initially, we introduce our research questions and the corpus of programs that we utilize for the assessment of WASM-MUTATE. Next, we elaborate on the methodology for each research question. For the sake of reproducibility, our data and experimenting pipeline are publicly available at `https://github.com/ASSERT-KTH/tawasco`. Our experiments are conducted in Standard F4s-v2(Skylake) Azure machines with 4 virtual cpus and 8GiB memory per instance.

RQ1: **To what extent are the program variants generated by** WASM-MUTATE **statically different from the original programs?** We check whether the WebAssembly binary variants radpidly produced by WASM-MUTATE are different from the original WebAssembly binary. Then, we assess whether the x86 machine code produced by wasmtime engine is also different.

RQ2: **How fast can** WASM-MUTATE **generate program variants that exhibit different execution traces?** To assess the versatility of WASM-MUTATE, we also examine the presence of different behaviors in the generated variants. Specifically, we measure the speed at which WASM-MUTATE generates variants with distinct machine code instruction traces and memory access patterns.

RQ3: **To what extent does** WASM-MUTATE **prevent side-channel attacks on WebAssembly programs?** Diversification being an option to prevent security issues, we assess the impact of WASM-MUTATE in preventing one class of attacks: cache attacks (Spectre).

4.1. Corpora

We answer our research questions with a corpus of 307 programs (303 + 4). These programs are summarized in Table 1. Each row in the table corresponds to the used programs, with the columns providing: where the program is sourced from, the number of programs, research question addressed, function count, the total number of instructions found in the original WebAssembly program and the type of attack that the original program was subjected to.

We answer RQ1 and RQ2 with corpus of programs from Cabrera et.al. [7], it is shown in the first row of Table 1. The corpus contains 303. The corpus contains programs for a range of tasks, from simple ones, such as sorting, to complex algorithms like a compiler lexer. The number of functions for each program ranges from 7 to 103 and, the number of total instructions ranges from 170 to 36023. All programs in corpus: 1) do not require input from user, i.e., do not functions like `scanf`, 2) terminate, 3) are deterministic, i.e., given the same input, provide the same output and 4) compile to WebAssembly using `wasi-clang` to compile them.

We answer RQ3 with four WebAssembly programs and three Spectre attack scenarios, from the Swivel project [36]. These programs are summarized in the final four rows of our corpus table. The first two programs are manually crafted and contain 16 functions, with instruction counts of 743 and 297, respectively. These binaries are specifically designed to perform the Spectre branch target attack. The third and fourth programs, documented in rows four and five, comefrom the Safeside project [20]. Unlike the first two, these binaries are significantly larger, each containing nearly 3000 functions and more than 300000 instructions. They are utilized for conducting the Spectre Return Stack (RSB) and Spectre Pattern History (PHT) attacks [28].

There is a notable difference in the number of functions and instructions between the first pair of Swivel binaries and the latter pair. This disparity can be attributed to the varying compilation processes applied to these WebAssembly binaries. The three attack scenarios are described in details in subsection 4.4.

4.2. Protocol for RQ1

With RQ1, we assess the ability of WASM-MUTATE to generate WebAssembly binaries that are different from the original program, including after their compilation to x86 machine code. In Figure 4 we show the steps we follow to answer RQ1. We run WASM-MUTATE on our corpus of 303 original C programs (step ① in figure). To generate the variants: 1) we start with one original and pass it to WASM-MUTATE to generate a variant; 2) the variant and the original program form a population of programs; 3) we randomly select a program from this population and pass it to WASM-MUTATE to generate a variant, which we add to the population; 4) we then restart the process in the previous step. to stack more mutations This procedure is carried out for a duration of 1 hour. The final outcome (step ② in figure) is a population with a number of stacked transformations, all starting from an original WebAssembly program. We then count the number of unique variants in the population. We compute the sha256 hash of each variant bytestream in order and define the population size metric as:

Metric 1. *Population_size(P): Given an original WebAssembly program P, a generated corpus of WebAssembly programs* $V = \{v_1, v_2, ..., v_N\}$ *where* v_i *is a variant of P, the population size is defined as:*

$$|set(\{sha256(v_1), ...sha256(v_N)\})| \ \forall v_i \in V$$

Since WebAssembly binaries may be further transformed into machine code before they execute, we also check that this additional transformations preserve the difference introduces by WASM-MUTATE in the WebAssembly binary. We use the wasmtime JIT compiler, cranelift, with all available optimizations, to generate the x86 binaries for each WebAssembly program and its variants (step ③ in figure). Then, we calculate the number of unique variants machine code representation for wasmtime. Counting the number of unique machine code, we compute the diversification preservation ratio:

Metric 2. *Ratio of preserved variants: Given an original WebAssembly program P and its population size as defined in Metric 1 and the JIT compiler C, we defined the ratio of preserved variants as:*

$$\frac{|set(\{sha256(C(v_1)), ...sha256(C(v_N))\})|}{Population\_size\ (P)} \ \forall v_i \in V$$

If $sha256(P_1) \neq sha256(P_2)$ and $sha256(C(P_1))) \neq sha256(C(P_2))$, this means that both programs are still different after being compiled to machine code, and this means that the cranelift compiler has not removed the transformations made by WASM-MUTATE.

Note that the protocol described earlier can be mapped to Algorithm 2. For instance, to measure population size for each tested program, one could measure how often the execution of Algorithm 2 reaches line 11. Similarly, to assess the level of preservation, one could track the frequency with which the algorithm arrives at line 13.

4.3. Protocol for RQ2

For RQ2, we evaluate how fast WASM-MUTATE can generate variants that offer distinct traces compared with the original program. We start by collecting the traces of the original program when executed in wasmtime. While continuously generating variants with random stacked transformations, we collect the execution traces of the variants as well. We record the time passed until we generate a variant that offers different execution traces, according to two types of traces: machine code instructions and memory accesses. This process can be seen in the enclosed square of Figure 4, annotated with RQ2.

We gather the instructions and memory traces utilizing IntelPIN [33, 16] (step ④ in the figure). To only collect the traces of the WebAssembly execution with a wasmtime engine, we pause and resume the collection as the execution leaves and re-enters the WebAssembly code, respectively. We implement this filtering with the built-in hooks of wasmtime. In addition, we disable ASLR on the machine where the variants are executed. This latter action ensures that the placement of the instructions in memory is deterministic. Examples of the traces we collect can be seen in Listing 7 and Listing 8 for memory and instruction traces, respectively.

```
[Writ] 0x555555ed1570 size=4 value=0x10dd0
[Read] 0x555555ed1570 size=4 value=0x10dd0
```

Listing 7: Memory trace with two events out of IntelPIN for the execution of a WebAssembly program with wasmtime. Trace events record: the type of the operation, read or write, the memory address, the number of bytes affected and the value read or written.

```
[I] mov rdx, qword ptr [r14+0x100]
[I] mov dword ptr [rdx+0xe64], ecx
```

Listing 8: Instructions trace with two events out of IntelPIN for the execution of a WebAssembly program with wasmtime. Each event records the corresponding machine code that executes.

In the text below, we outline the metric used to assess how fast WASM-MUTATE can generate variants that provide different execution traces.

Metric 3. *Time until different trace: Given an original WebAssembly program P, and an its execution trace* T_1*, the time until different trace is defined as the time between the diversification process starts and the when the variant V is generated with execution trace* T_2*, and* $T_1 \neq T_2$*.*

Notice that the previously defined metric is instantiated twice, for instructions and memory type of events.

Refering to Algorithm 2, we quantify the elapsed time between line 6 and line 16 to obtain the time it takes for WASM-MUTATE to generate a unique WebAssembly variant producing different execution traces.

4.4. Protocol for RQ3

To answer RQ3, we apply WASM-MUTATE to the same security WebAssembly programs used by Narayan et al. to evaluated Swivel's ability at protecting WebAssembly programs against side-channel attacks [36]. The four cache timing side-channel attacks are presented in detail in subsection 4.1. The specific binary and its corresponding attack can be appreciated in Table 1. We evaluate to what extent WASM-MUTATE can prevent such attacks. In the following text, we describe the attacks we replicate and evaluate in order of answering RQ3.

Narayan and colleagues successfully bypass the control flow integrity safeguards, using speculative code execution as detailed in [28]. Thus, we use the same three Spectre attacks from Swivel: 1) The Spectre Branch Target Buffer (btb) attack exploits the branch target buffer by predicting the target of an indirect jump, thereby rerouting speculative control flow to an arbitrary target. 2) The Spectre Pattern History Table (pht) takes advantage of the pattern history table to anticipate the direction of a conditional branch during the ongoing evaluation of a condition. 3) The Spectre Return Stack Buffer (ret2spec) attack exploits the return stack buffer that stores the locations of recently executed call instructions to predict the target of `ret` instructions. Each attack methodology relies on the extraction of memory bytes from another hosted WebAssembly binary that executes in parallel.

For each of the four WebAssembly binaries introduced in subsection 4.1, we generated a maximum of 1000 random stacked transformations utilizing 100 distinct seeds. This resulted in a total of 100,000 variants for each original WebAssembly binary. We then assess the success rate of attacks across these variants by measuring the bandwidth of the exfiltrated data, that is: the rate of correctly leaked bytes per unit of time. We then count the correctly exfiltrated bytes and divided them by the variant program's execution time.

Notice that, the bandwidth metric captures not only whether the attacks are successful or not, but also the degree to which the data exfiltration is hindered. For instance, a variant that continues to exfiltrate secret data but does so over an impractical duration would be deemed as having been hardened. For this, we state the bandwidth metric in the following definition :

Metric 4. *Bandwidth: Given data* $D = \{b_0, b1, ..., b_C\}$ *being exfiltrated in time* T *and* $K = k_1, k_2, ..., k_N$ *the collection of correct data bytes, the bandwidth metric is defined as:*

$$\frac{|b_i \text{ such that } b_i \in K|}{T}$$

5. Experimental Results

5.1. To what extent are the program variants generated by WASM-MUTATE statically different from the original programs?

To address RQ1, we utilize WASM-MUTATE to process the original 303 programs from [7]. WASM-MUTATE is set to generate variants with a timeout of one hour for each individual program. Following this, we assess the sizes of their variant populations as well as their corresponding preservation ratio (Refer to Metric 1 and Metric 2 for more details).

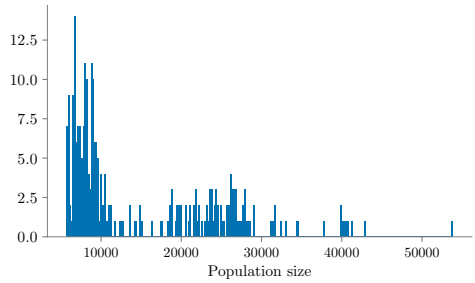


Figure 5: RQ1: Number of unique WebAssembly programs generated by WASM-MUTATE in 1 hour out of each program of the corpus.

In Figure 5, we show the distribution of the population size generated out of WASM-MUTATE. WASM-MUTATE successfully diversifies all 303 original programs, yielding a diversification rate of 100%. Within an hour, WASM-MUTATE demonstrates its impressive efficiency and effectiveness by producing a median of 9500 unique variants for the 303 original programs. The largest population size observed is 53816, while the smallest is 5716. There are several factors contributing to large population sizes.

WASM-MUTATE can diversify functions within WASI-libc. Despite the relatively low function count in the original source code, WASM-MUTATE creates thousands of distinct variants in the function of the incorporated libraries. This feature improves over methods that can only diversify the original source code processed through the LLVM compilation pipeline [7].

We have observed a significant variation in the population size out of WASM-MUTATE between different programs, ranging by several thousand variants (from a maximum of 53816 variants to a minimum of 5716 variants). This disparity is attributed to: the non-deterministic nature of WASM-MUTATE and 2) the characteristics of the program. WASM-MUTATE mutates a randomly selected portion of a program. If the selected instruction is determined to be non-deterministic, despite the transformation being semantically equivalent, WASM-MUTATE discards the variant and moves on to another random transformation. For instance, if the instruction targeted for mutation is a function call, WASM-MUTATE proceeds to the next one. This process, in conjunction with the unique characteristics of each program, results in a varying population size. For example, an input binary with a high number of function calls would lead to a greater number of trials and errors, slowing down the generation of variants, thereby resulting in a smaller overall population size for 1 hour of WASM-MUTATE execution.

As stated in subsection 4.2, we also assess static diver-

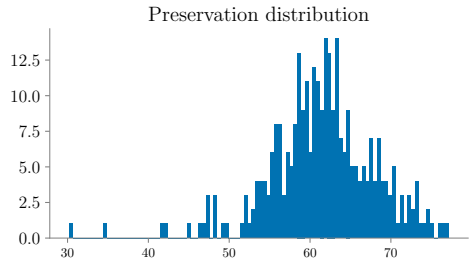


Figure 6: RQ1: Distribution of the ratio of wasmtime preserved variants.

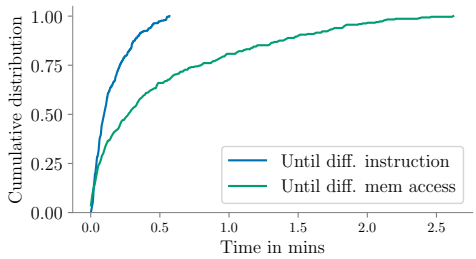


Figure 7: RQ2: Cumulative distribution for time until different trace. In blue for different machine code instructions, in green for different memory traces. The X-axis marks time in minutes, and the Y-axis shows the ratio of programs from 303 for which WASM-MUTATE created a variant within that time.

sification with Metric 2 by calculating the preservation ratio of variant populations. Figure 6 presents the distribution of preservation ratios for the cranelift compiler of wasmtime. We have observed a median preservation ratio of 62%. On the one hand, we have observed that there is no correlation between population size and preservation ratio. In other words, having a larger population size does not necessarily lead to a higher preservation ratio. On the other hand, the phenomena of non-preserved variants can be explained as follows. Factors such as custom sections are often disregarded by compilers. Similarly, bloated code plays a role in this context. For instance, WASM-MUTATE generates certain variants with unused types or functions, which are then detected and eliminated by cranelift. Yet, note that even when working with the smallest population size and the lowest preservation percentage, the number of unique machine codes can still encompass thousands of variants.

Answer to RQ1: WASM-MUTATE generates WebAssembly variants for the 303 programs, which are different from the original program. Within a one-hour diversification budget, WASM-MUTATE synthesizes more than 9000 unique variants per program on average. 62% of the variants remain different after machine-code compilation. WASM-MUTATE is good at producing a large number of WebAssembly program variants.

5.2. How fast can WASM-MUTATE generate program variants that exhibit different execution traces?

To answer question RQ2, we measure how long it takes to generate one variant that exhibits execution traces that are different from the original. In Figure 7, we display a cumulative distribution plot showing the time required for WASM-MUTATE to generate variants with different traces, in blue for machine code instructions and green for memory traces. The X-axis marks time in minutes, and the Y-axis shows the ratio of programs from 303 for which WASM-MUTATE created a variant within that time. For all original program, WASM-MUTATE succeeds in generating one variant with different traces comparing to the original program, either in machine code instructions or memory access, ie both cumlative distributions readch 100The shortest time to generate a variant with different machine code instruction traces is 0.12 seconds, and for different memory traces, it is 0.06 seconds. In the slowest scenarios, WASM-MUTATE takes under 1 minute for different machine code instruction traces and less than 3 minutes for different memory traces. Overall, WASM-MUTATE takes a median of 5.4 seconds and 12.6 seconds in generating variants with different machine code instructions and different memory instructions respectively.

The use an e-graph random traversal is the key factor for such a fast generation process. Once WASM-MUTATE locates a modifiable instruction within the binary and constructs its corresponding e-graph, traversal is virtually instantaneous. However, the time efficiency of variant generation is not consistent across all programs, as illustrated in Figure 7. This variation primarily stems from the varying complexities of the programs under analysis, as previously mentioned in subsection 5.1. Interestingly, WASM-MUTATE may attempt to build e-graphs from instructions that, while not inherently leading to undefined behavior, are part of a data flow graph that could. For example, the data flow graph might be dependent on a function call. Although transforming undefined behavioural instructions is deactivated by default in WASM-MUTATE to maintain functional equivalence with the original code, the process of attempting to construct such e-graphs can extend the duration of the diversification pass. As a result, WASM-MUTATE may require multiple attempts to successfully create and traverse an e-graph, impacting the rate at which it generates behaviorally distinct variants. This phenomenon is particularly noticeable in original programs that have a high frequency of function calls.

In average, WASM-MUTATE takes three times longer to synthesize unique memory traces than it does to generate different instruction traces (as it can be observed in how the green plot of the figure is skewed to the right). The main reason for this difference is the limited set of rewriting rules that specifically focus on memory operations. WASM-

MUTATE includes more rules for manipulating code, which increases the odds of generating a variant with diverse machine code instructions. Additionally, the variant creation process halts and restarts with alternative rewriting rules if WASM-MUTATE detects that the selected code for transformation could result in unpredictable behavior.

We have identified four primary factors explaining why execution traces differs overall. First, alterations to the binary layout inherently impact both machine code instruction traces and memory accesses within the program's stack. In particular, WASM-MUTATE creates variants that change the return addresses of functions, leading to divergent execution traces, including those related to memory access. Second, our rewriting rules incorporate artificial global values into WebAssembly binaries. Since these global variables are inherently manipulated via the stack, their access inevitably generate divergent memory traces. Third, WASM-MUTATE injects 'phantom' instructions which do not aim to modify the outcome of a transformed function during execution. These intermediate calculations trigger the spill/reload component of the runtime, varying spill and reload operations. In the context of limited physical resources, these operations temporarily store values in memory for later retrieval and use, thus creating unique memory traces. Finally, certain rewriting rules implemented by WASM-MUTATE replicate fragments of code, e.g., performing commutative operations. These code segments may contain memory accesses, and while neither the memory addresses nor their values change, the frequency of these operations does. Overall, these findings influence the diversity of execution traces among the generated variants.

Answer to RQ2: WASM-MUTATE generates variants with distinct machine code instructions and memory traces for all tested programs. The quickest time for generating a variant with a unique machine code trace is 0.12 seconds, and for divergent memory traces, it's best time is 0.06 seconds. On average, the median time required to produce a variant with distcint traces stands at 5.4 seconds for unique machine code traces and 16.2 seconds for different memory traces. These metrics indicate that WASM-MUTATE is suitable for fast-moving target defense strategies, capable of generating a new variant in well under a minute [6]. To the best of our knowledge, WASM-MUTATE is the fastest diversification engine for WebAssembly.

5.3. To what extent does WASM-MUTATE prevent side-channel attacks on WebAssembly programs?

To answer RQ3, we execute WASM-MUTATE on four distinct binaries WebAssembly susceptible to Spectre related attacks. Each of the four programs is transformed with one of for 100 different seeds and up to 1000 stacked transformations. We assess the resulting impact of the attacks as outlined in 4.4. The analysis encompasses a total of 4×100×1000 binaries, which also includes the original four.

Figure 8 offers a graphical representation of WASM-MUTATE's influence on the Swivel original programs and their attacks. Each plot corresponds to one original WebAssembly binary and the attack it undergoes: btb_breakout, btb_leakage, ret2spec, and pht. The Y-axis represents the exfiltration bandwidth (see Metric 4). The bandwidth of the original binary under attack is marked as a blue dashed horizontal line. In each plot, the variants are grouped in clusters of 100 stacked transformations. These are indicated by green dots and lines. The dot signifies the median bandwidth for the cluster, while the line represents the interquartile range of the group's bandwidth.

For btb_breakout and btb_leakage, WASM-MUTATE demonstrates effectiveness, generating variants that leak less information than the original in 78% and 70% of the cases, respectively. For these particular binaries, a significant reduction in exfiltration bandwidth to zero is noted after 200 stacked transformations. This means that with a minimum of 200 stacked transformations, WASM-MUTATE can create variants that are completely resistant to the original attack. For the ret2spec and pht scenarios, the produced variants consistently exhibit lower bandwidth than the original in 76% and 71% of instances, respectively. As depicted in the plots, the exfiltration bandwidth diminishes following the application of at least 100 stacked transformations.

This success is explained by the fact that WASM-MUTATE synthesizes variants that effectively alter memory access patterns. Specifically, it does so by amplifying spill/reload operations, injecting artificial global variables, and changing the frequency of pre-existing memory accesses. These transformations influence the WebAssembly program's memory, causing disruption to cache predictors. As a result, these alterations contribute to a reduction in exfiltration bandwidth.

Furthermore, many attacks rely on a timer component to measure cache access time for memory, and disrupting this component effectively impairs the attack's effectiveness. This strategy of dynamic alteration has also been employed in other scenarios. For instance, to counter potential timing attacks, Firefox randomizes its built-in JavaScript timer [42]. WASM-MUTATE applies the same strategy by interspersing instructions within the timing steps of WebAssembly variants. In Listing 9 and Listing 10, we demonstrate WASM-MUTATE's impact on time measurements. The former illustrates the original time measurement, while the latter presents a variant with WASM-MUTATE-inserted operations amid the timing.

```
;; Code from original btb_breakout
...
(call $readTimer)
(set_local $end_time)
... access to mem
(i64.sub (get_local $end_time ) (get_local $start_time))
(set_local $duration)
...
```

Listing 9: Wasm timer used in btb_breakout program.

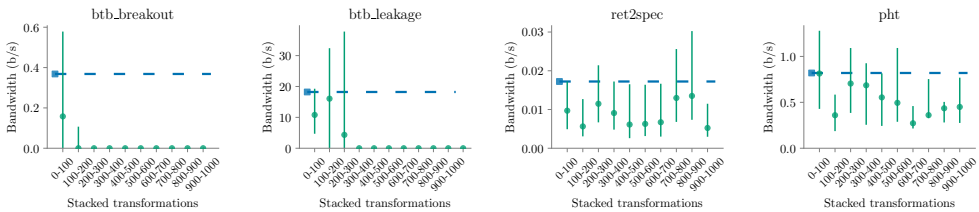


Figure 8: Visual representation of WASM-MUTATE's impact on Swivel's original programs. The Y-axis denotes exfiltration bandwidth, with the original binary's bandwidth under attack highlighted by a blue marker and dashed line. Variants are clustered in groups of 100 stacked transformations, denoted by green dots (median bandwidth) and lines (interquartile bandwidth range). Overall, for all 100000 variants generated out of each original program, 70% have less data leakage bandwidth.

```
;; Variant code
...
(call $readTimer)
(set_local $end_time)
<inserted instructions>
... access to mem
<inserted instructions>
(i64.sub (get_local $end_time ) (get_local $start_time))
(set_local $duration)
...
```

Listing 10: Variant of btb_breakout with more instructions added in between time measurement.

WASM-MUTATE proves effective against cache access timers because the time measurement of single or a few instructions is inherently different. By introducing more instructions, this randomness is amplified, thereby reducing the timer's accuracy.

Furthermore, CPUs have a maximum capacity for the number of instructions they can cache. WASM-MUTATE injects instructions in such a way that the vulnerable instruction may exceed this cacheable instruction limit, meaning that caching becomes disabled. This kind of transformation can be viewed as padding [15]. In Listing 11 and Listing 12, we illustrate the effect of WASM-MUTATE on padding instructions. Listing 11 presents the original code used for training the branch predictor, along with the expected speculated code.

```
;; Code from original btb_breakout
...
;; train the code to jump here (index 1)
(i32.load (i32.const 2000))
(i32.store (i32.const 83)) ;; just prevent optimization
...
;; transiently jump here
(i32.load (i32.const 339968)) ;; S(83) is the secret
(i32.store (i32.const 83)) ;; just prevent optimization
```

Listing 11: Two jump locations in btb_breakout. The top one trains the branch predictor, the bottom one is the expected jump that exfiltrates the memory access.

```
;; Variant code
...
;; train the code to jump here (index 1)
<inserted instructions>
(i32.load (i32.const 2000))
<inserted instructions>
(i32.store (i32.const 83)) ;; just prevent optimization
...
;; transiently jump here
<inserted instructions>
(i32.load (i32.const 339968)) ;; "S"(83) is the secret
<inserted instructions>
(i32.store (i32.const 83)) ;; just prevent optimization
...
```

Listing 12: Variant of btb_breakout with more instructions added indindinctly between jump places.

The padding alters the arrangement of the binary code in memory, effectively impeding the attacker's capacity to initiate speculative execution. Even when an attack is launched and the vulnerable code is "speculated", the memory access is not impacted as planned.

In every program, we note that the exfiltration bandwidth tends to be greater than the original when the variants include a small number of transformations. This indicates that, although the transformations generally contribute to the reduction of data leakage, the initial few might not consistently contribute positively towards this objective. We have identified several fundamental reasons, which we discuss below.

Firstly, as emphasized in prior applications of WASM-MUTATE [8], uncontrolled diversification can be counterproductive if a specific objective, such as a cost function, is not established at the beginning of the diversification process. Secondly, while some transformations yield distinct WebAssembly binaries, their compilation produces identical machine code. Transformations that are not preserved undermine the effectiveness of diversification. For example, incorporating random `nop` operations directly into WebAssembly does not modify the final machine code as the `nop` operations are often removed by the compiler. The same phenomenon is observed with transformations to custom sections of WebAssembly binaries. Additionally, it is impor-

tant to note that transformed code doesn't always execute, i.e., WASM-MUTATE may generate dead code.

Finally, for ret2spec and pht, both programs are hardened with attack bandwidth reduction, but this does not materialize in a short-term timeframe (low count of stacked transformations). Furthermore, the exfiltration bandwidth is more dispersed for these two programs. Our analysis indicates a correlation between bandwidth reduction and the complexity of the binary subject to diversification. Ret2spec and pht are considerably larger than btb_breakout and btb_leakage. The former comprises more than 300k instructions, while the latter two include fewer than 800 instructions. Given that WASM-MUTATE applies precise, fine-grained transformations one at a time, the likelihood of impacting critical attack components, such as timing memory accesses, diminishes for larger binaries, particularly when limited to 1,000 transformations. Based on these observations, we believe that a greater number of stacked transformations would further contribute to eventually eliminating the attacks associated with ret2spec and pht.

Answer to RQ3: software diversification is effective at synthesizing WebAssembly binaries that are less susceptible to Spectre-like attacks. WASM-MUTATE generates variants of btb_breakout and btb_leakage that are totally protected and hardened variants of ret2spec and pht that are more resilient than the original program. In total, 70% of the diversified variants exhibit a reduced effectiveness (reduced data leakage bandwidth) compared to the original program. Larger programs require a greater number of transformations to effectively neutralize the attacks.

6. Discussion

Fuzzing WebAssembly compilers with WASM-MUTATE In fuzzing campaigns, selecting the appropriate starting inputs is both a significant challenge and essential for detecting bugs promptly [46]. This is particularly true with compilers, where the inputs should be well-formed yet intricate enough programs to probe various compiler components. WASM-MUTATE could address this challenge by generating semantically equivalent variants from an original WebAssembly binary, enhancing the scope and efficiency of the testing process. A practical example of this occurred in 2021, when this approach led to the discovery of a wasmtime security CVE [18]. Through the creation of semantically equivalent variants, the spill/reload component of cranelift was stressed, resulting in the discovering and subsequent resolution of the before-mentioned CVE.

Mitigating Port Contention Rokicki et al. [39] showed the practicality of a covert side-channel attack using port contention within WebAssembly code in the browser. This attack fundamentally relies on the precise prediction of Wasm instructions that trigger port contention on specific ports. To combat this security concern, we propose an man-in-the-middle mechanism, WASM-MUTATE, which could be conveniently implemented as a browser plugin. WASM-MUTATE has the ability to replace the WebAssembly instructions used as port contention predictor with other random instructions. This will inevitably remove the port contention in the specific port used to conduct the attack, hardening browsers against such exploitative maneuvers.

7. Related Work

Static software diversification refers to the process of synthesizing, and distributing unique but functionally equivalent programs to end users. The implementation of this process can take place at any stage of software development and deployment - from the inception of source code, through the compilation phase, to the execution of the final binary [24, 34]. WASM-MUTATE, a static diversifier, can be placed at the final stage, keeping in mind that the code will subsequently undergo final compilation by JIT compilers. The concept of software diversification owes much to the pioneering work of Cohen [12]. His suite of code transformations aimed to increase complexity and thereby enhance the difficulty of executing a successful attack against a broad user base [12]. WASM-MUTATE's rewriting rules draw significantly from Cohen and Forrest seminal contributions [12, 19].

Jackson and colleagues [24] proposed that the compiler can play a pivotal role in promoting static software diversification. In the context of WebAssembly, CROW leverages compiler technology for diversification. It is a superdiversifier [25],for WebAssembly that is built in the LLVM compilation tool chain. However, integrating the diversifier directly into the LLVM compiler, restricts the tool's applicability to WebAssembly binaries generated through LLVM. This implies that any WebAssembly source code that lacks an LLVM frontend implementation cannot take advantage of CROW's capabilities. In contrast, WASM-MUTATE provides a more versatile and faster WebAssembly to WebAssembly diversification solution, maintaining compatibility with any compiler. Secondly, unlike CROW, WASM-MUTATE does not rely on an SMT solver to validate the generated variants. Instead, it guarantees semantic equivalence by design, resulting in greater efficiency in generating WebAssembly variants, as discussed in subsection 5.1. As a WebAssembly to WebAssembly diversification tool, WASM-MUTATE augments the range of tools capable of generating WebAssembly programs, a topic explored comprehensively throughout this work.

The process of diversifying a WebAssembly program can be conceptualized as a three-stage procedure: parsing the program, transforming it, and finally re-encoding it back into WebAssembly. Our review of the literature has revealed several studies that have employed parsing and encoding components for WebAssembly binaries across various domains. This indicates that these works accept a WebAssembly binary as an input and output a unique WebAssembly binary. These domains span optimization [47], control flow [2], and dynamic analysis [31, 43, 2, 3]. When the transformation stage introduces randomized mutations to

the original program, the aforementioned tools could potentially be construed as diversifiers. WASM-MUTATE is related to these previous works, as it can serve as an optimizer or a test case reducer due to the incorporation of an e-graph at the heart of its diversification process [44]. To the best of our knowledge, the introduction of an e-graph into WASM-MUTATE marks the first endeavor to integrate an e-graph into a WebAssembly to WebAssembly analysis tool.

BREWasm [10] offers a comprehensive static binary rewriting framework for WebAssembly and can be considered to be the most similar to WASM-MUTATE. For instance, it can be used to model a diversification engine. It parses a Wasm binary into objects, rewrites them using fine-grained APIs, integrates these APIs to provide high-level ones, and re-encodes the updated objects back into a valid Wasm binary. The effectiveness and efficiency of BREWasm have been demonstrated through various Wasm applications and case studies on code obfuscation, software testing, program repair, and software optimization. The implementation of BREWasm follows a completely different technical approach. In comparison with our work, the authors pointed out that our tool employs lazy parsing of Wasm. Although they perceived this as a limitation, it is eagerly implemented to accelerate the generation of WebAssembly binaries. Additionally, our tool leverages the parser and encoder of wasmtime, a standalone compiler and interpreter for Wasm, thereby boosting its reliability and lowering its error-prone nature.

Another similar work to WASM-MUTATE is WASMixer [11]. WASMixer focuses on three code obfuscation methods for WebAssembly binaries: memory access encryption, control flow flattening, and the insertion of opaque predicates. Their strategy is specifically designed for obfuscating Wasm binaries. In contrast, while WASM-MUTATE does not employ memory access encryption or control flow flattening, it can still function effectively as an obfuscator. Previous evaluations confirm that WASM-MUTATE has been successful in evading malware detection [8]. On the same topic, Madvex [32] also aims to modify Wasm binaries to achieve malware evasion, but their approach is principally driven by a generic reward function and is largely confined to altering only the code section of a Wasm binary. WASM-MUTATE, however, adopts a more flexible strategy by applying a broader array of transformations, which are not limited to the code section. Consequently, WASM-MUTATE is capable of generating malware variants without negatively affecting either their code or performance.

8. Conclusion

WASM-MUTATE is a fast and effective diversification tool for WebAssembly, with a 100% diversification rate across the 303 programs of the considered benchmark. With respect to speed, it creates over 9000 unique variants per hour. The WASM-MUTATE workflow ensures that all final variants offer different and unique execution traces. We have proven that WASM-MUTATE is able to mitigate Spectre attacks in WebAssembly, producing fully protected variants of two versions of the btb attack, and variants of ret2spec and pht that leak less data than the original ones.

In future work, we aim to fine-tune the diversification process, balancing broad diversification with the needs of specific scenarios. Besides, the creation of rewriting rules for WASM-MUTATE is currently a manual task, yet we have identified potential for automation. For instance, WASM-MUTATE could be enhanced through data-driven methods such as rule mining. Furthermore, we have observed that the impact of WASM-MUTATE on ret2spec and pht attacks is considerably less compared to btb attacks. These attacks exploit the returning address of executed functions in the program stack. One mitigation of this would be multi-variant execution strategy, implemented on top of WASM-MUTATE. By offering different execution paths, the returning addresses on the stack at each function execution would vary, thereby improving the hardening of binaries against ret2spec attacks.

References

[1] Azad, B.A., Laperdrix, P., Nikiforakis, N., 2019. Less is more: Quantifying the security benefits of debloating web applications, in: 28th USENIX Security Symposium (USENIX Security 19), USENIX Association, Santa Clara, CA. pp. 1697–1714. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/azad.

[2] Breitfelder, F., Roth, T., Baumgärtner, L., Mezini, M., 2023. Wasma: A static webassembly analysis framework for everyone, in: 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 753–757. doi:10.1109/SANER56733.2023.00085.

[3] Brito, T., Lopes, P., Santos, N., Santos, J.F., 2022. Wasmati: An efficient static vulnerability scanner for webassembly. Computers & Security 118, 102745. URL: https://www.sciencedirect.com/science/article/pii/S0167404822001407, doi:https://doi.org/10.1016/j.cose.2022.102745.

[4] Bruschi, D., Cavallaro, L., Lanzi, A., 2007. Diversified process replicæ for defeating memory error exploits, in: 2007 IEEE International Performance, Computing, and Communications Conference, pp. 434–441. doi:10.1109/PCCC.2007.358924.

[5] Cabrera-Arteaga, J., Donde, S., Gu, J., Floros, O., Satabin, L., Baudry, B., Monperrus, M., 2020. Superoptimization of webassembly bytecode, in: Proceedings of MoreVMs: Workshop on Modern Language Runtimes. URL: http://arxiv.org/pdf/2002.10213, doi:10.1145/3397537.3397567.

[6] Cabrera Arteaga, J., Laperdrix, P., Monperrus, M., Baudry, B., 2022. Multi-variant execution at the edge, in: Proceedings of the 9th ACM Workshop on Moving Target Defense, Association for Computing Machinery, New York, NY, USA. p. 11–22. URL: https://doi.org/10.1145/3560828.3564007, doi:10.1145/3560828.3564007.

[7] Cabrera Arteaga, J., Malivitsis, O.F., Pérez, O.L.V., Baudry, B., Monperrus, M., 2021. Crow: Code diversification for webassembly. URL: https://madweb.work/preprints/madweb21-paper4-pre_print_version.pdf, arXiv:2008.07185.

[8] Cabrera-Arteaga, J., Monperrus, M., Toady, T., Baudry, B., 2023. Webassembly diversification for malware evasion. Computers & Security 131, 103296. URL: https://www.sciencedirect.com/science/article/pii/S0167404823002067, doi:https://doi.org/10.1016/j.cose.2023.103296.

[9] Cao, D., Kunkel, R., Nandi, C., Willsey, M., Tatlock, Z., Polikarpova, N., 2023. Babble: Learning better abstractions with e-graphs and anti-unification. Proc. ACM Program. Lang. 7. URL: https://doi.org/10.1145/3571207, doi:10.1145/3571207.

[10] Cao, S., He, N., Guo, Y., Wang, H., 2023a. A General Static

Binary Rewriting Framework for WebAssembly. arXiv e-prints , arXiv:2305.01454doi:10.48550/arXiv.2305.01454, arXiv:2305.01454.

[11] Cao, S., He, N., Guo, Y., Wang, H., 2023b. WASMixer: Binary Obfuscation for WebAssembly. arXiv e-prints , arXiv:2308.03123doi:10.48550/arXiv.2308.03123, arXiv:2308.03123.

[12] Cohen, F.B., 1993. Operating system protection through program evolution. Computers & Security 12, 565–584.

[13] Crane, S.J., Volckaert, S., Schuster, F., Liebchen, C., Larsen, P., Davi, L., Sadeghi, A.R., Holz, T., De Sutter, B., Franz, M., 2015. It's a trap: Table randomization and protection against function-reuse attacks, in: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Association for Computing Machinery, New York, NY, USA. p. 243–255. URL: https://doi.org/10.1145/2810103.2813682, doi:10.1145/2810103.2813682.

[14] Dongarra, J.J., Hinds, A., 1979. Unrolling loops in fortran. Software: Practice and Experience 9, 219–226.

[15] Duck, G.J., Gao, X., Roychoudhury, A., 2020. Binary rewriting without control flow recovery, in: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, Association for Computing Machinery, New York, NY, USA. p. 151–163. URL: https://doi.org/10.1145/3385412.3385972, doi:10.1145/3385412.3385972.

[16] D'Elia, D.C., Invidia, L., Palmaro, F., Querzoni, L., 2022. Evaluating dynamic binary instrumentation systems for conspicuous features and artifacts. Digital Threats 3. URL: https://doi.org/10.1145/3478520, doi:10.1145/3478520.

[17] Fastly, 2020. The power of serverless, 72 times over. URL: https://www.fastly.com/blog/the-power-of-serverless-at-the-edge.

[18] Fastly, 2021. Stop a wasm compiler bug before it becomes a problem | fastly. https://www.fastly.com/blog/defense-in-depth-stopping-a-wasm-compiler-bug-before-it-became-a-problem.

[19] Forrest, S., Somayaji, A., Ackley, D., 1997. Building diverse computer systems, in: Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No.97TB100133), pp. 67–72. doi:10.1109/HOTOS.1997.595185.

[20] Google, 2020. Safeside. https://github.com/PLSysSec/safeside. URL: https://github.com/PLSysSec/safeside.

[21] Haas, A., Rossberg, A., Schuff, D.L., Titzer, B.L., Holman, M., Gohman, D., Wagner, L., Zakai, A., Bastien, J., 2017a. Bringing the web up to speed with WebAssembly, in: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 185–200.

[22] Haas, A., Rossberg, A., Schuff, D.L., Titzer, B.L., Holman, M., Gohman, D., Wagner, L., Zakai, A., Bastien, J., 2017b. Bringing the web up to speed with webassembly, in: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, Association for Computing Machinery, New York, NY, USA. p. 185–200. URL: https://doi.org/10.1145/3062341.3062363, doi:10.1145/3062341.3062363.

[23] Hilbig, A., Lehmann, D., Pradel, M., 2021. An empirical study of real-world webassembly binaries: Security, languages, use cases, in: Proceedings of the Web Conference 2021, pp. 2696–2708.

[24] Jackson, T., Salamat, B., Homescu, A., Manivannan, K., Wagner, G., Gal, A., Brunthaler, S., Wimmer, C., Franz, M., 2011. Compiler-generated software diversity, in: Moving Target Defense. Springer, pp. 77–98.

[25] Jacob, M., Jakubowski, M.H., Naldurg, P., Saw, C.W.N., Venkatesan, R., 2008. The superdiversifier: Peephole individualization for software protection, in: International Workshop on Security, Springer. pp. 100–120.

[26] Jetbrain, 2023. Kotlin wasm. https://kotlinlang.org/docs/wasm-overview.html. URL: https://kotlinlang.org/docs/wasm-overview.html.

[27] Kim, M., Jang, H., Shin, Y., 2022. Avengers, assemble! survey of webassembly security solutions, in: 2022 IEEE 15th International Conference on Cloud Computing (CLOUD), pp. 543–553. doi:10.1109/CLOUD55607.2022.00077.

[28] Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y., 2019. Spectre attacks: Exploiting speculative execution, in: 2019 IEEE Symposium on Security and Privacy (SP), pp. 1–19. doi:10.1109/SP.2019.00002.

[29] Koppel, J., Guo, Z., de Vries, E., Solar-Lezama, A., Polikarpova, N., 2022. Searching entangled program spaces. Proc. ACM Program. Lang. 6. URL: https://doi.org/10.1145/3547622, doi:10.1145/3547622.

[30] Le, V., Afshari, M., Su, Z., 2014. Compiler validation via equivalence modulo inputs, in: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, Association for Computing Machinery, New York, NY, USA. p. 216–226. URL: https://doi.org/10.1145/2594291.2594334, doi:10.1145/2594291.2594334.

[31] Lehmann, D., Pradel, M., 2019. Wasabi: A framework for dynamically analyzing webassembly, in: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, Association for Computing Machinery, New York, NY, USA. p. 1045–1058. URL: https://doi.org/10.1145/3297858.3304068, doi:10.1145/3297858.3304068.

[32] Loose, N., Mächtle, F., Pott, C., Bezsmertnyi, V., Eisenbarth, T., 2023. Madvex: Instrumentation-based Adversarial Attacks on Machine Learning Malware Detection. arXiv e-prints , arXiv:2305.02559doi:10.48550/arXiv.2305.02559, arXiv:2305.02559.

[33] Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K., 2005. Pin: building customized program analysis tools with dynamic instrumentation. Acm sigplan notices 40, 190–200.

[34] Lundquist, G.R., Mohan, V., Hamlen, K.W., 2016. Searching for software diversity: attaining artificial diversity through program synthesis, in: Proceedings of the 2016 New Security Paradigms Workshop, pp. 80–91.

[35] Nandi, C., Willsey, M., Anderson, A., Wilcox, J.R., Darulova, E., Grossman, D., Tatlock, Z., 2020. Synthesizing structured cad models with equality saturation and inverse transformations, in: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, Association for Computing Machinery, New York, NY, USA. p. 31–44. URL: https://doi.org/10.1145/3385412.3386012, doi:10.1145/3385412.3386012.

[36] Narayan, S., Disselkoen, C., Moghimi, D., Cauligi, S., Johnson, E., Gang, Z., Vahldiek-Oberwagner, A., Sahita, R., Shacham, H., Tullsen, D., Stefan, D., 2021. Swivel: Hardening WebAssembly against spectre, in: 30th USENIX Security Symposium (USENIX Security 21), USENIX Association. pp. 1433–1450. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/narayan.

[37] Premtoon, V., Koppel, J., Solar-Lezama, A., 2020. Semantic code search via equational reasoning, in: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, Association for Computing Machinery, New York, NY, USA. p. 1066–1082. URL: https://doi.org/10.1145/3385412.3386001, doi:10.1145/3385412.3386001.

[38] Ren, X., Ho, M., Ming, J., Lei, Y., Li, L., 2021. Unleashing the hidden power of compiler optimization on binary code difference: An empirical study, in: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Association for Computing Machinery, New York, NY, USA. p. 142–157. URL: https://doi.org/10.1145/3453483.3454035, doi:10.1145/3453483.3454035.

[39] Rokicki, T., Maurice, C., Botvinnik, M., Oren, Y., 2022. Port contention goes portable: Port contention side channels in web browsers, in: Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security, Association for Computing Machinery, New York, NY, USA. p. 1182–1194. URL: https://doi.org/10.1145/3488932.3517411, doi:10.1145/3488932.3517411.

[40] Rossberg, A., 2019. WebAssembly Core Specification. Technical Report. W3C. URL: https://www.w3.org/TR/wasm-core-1/.

[41] Sasnauskas, R., Chen, Y., Collingbourne, P., Ketema, J., Lup, G., Taneja, J., Regehr, J., 2017. Souper: A Synthesizing Superopti-

mizer. arXiv e-prints , arXiv:1711.04422doi:10.48550/arXiv.1711.04422, arXiv:1711.04422.

[42] Schwarz, M., Maurice, C., Gruss, D., Mangard, S., 2017. Fantastic timers and where to find them: High-resolution microarchitectural attacks in javascript, in: Kiayias, A. (Ed.), Financial Cryptography and Data Security, Springer International Publishing, Cham. pp. 247–267.

[43] Stiévenart, Q., De Roover, C., 2020. Compositional information flow analysis for webassembly programs, in: 2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM), IEEE. pp. 13–24.

[44] Tate, R., Stepp, M., Tatlock, Z., Lerner, S., 2009. Equality saturation: A new approach to optimization, in: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Association for Computing Machinery, New York, NY, USA. p. 264–276. URL: https://doi.org/10.1145/1480881.1480915, doi:10.1145/1480881.1480915.

[45] Wagner, L., Mayer, M., Marino, A., Soldani Nezhad, A., Zwaan, H., Malavolta, I., 2023. On the energy consumption and performance of webassembly binaries across programming languages and runtimes in iot, in: Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering, Association for Computing Machinery, New York, NY, USA. p. 72–82. URL: https://doi.org/10.1145/3593434.3593454, doi:10.1145/3593434.3593454.

[46] Wang, J., Chen, B., Wei, L., Liu, Y., 2017. Skyfire: Data-driven seed generation for fuzzing, in: 2017 IEEE Symposium on Security and Privacy (SP), pp. 579–594. doi:10.1109/SP.2017.23.

[47] Wen, E., Dietrich, J., 2023. Wasmslim: Optimizing webassembly binary distribution via automatic module splitting, in: 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 673–677. doi:10.1109/SANER56733.2023.00069.

[48] Willsey, M., Nandi, C., Wang, Y.R., Flatt, O., Tatlock, Z., Panchekha, P., 2021. Egg: Fast and extensible equality saturation. Proc. ACM Program. Lang. 5. URL: https://doi.org/10.1145/3434304, doi:10.1145/3434304.

SCALABLE COMPARISON OF JAVASCRIPT V8 BYTECODE TRACES

Javier Cabrera-Arteaga, Martin Monperrus, Benoit Baudry
11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (SPLASH 2019)

https://doi.org/10.1145/3358504.3361228

Scalable Comparison of JavaScript V8 Bytecode Traces

Javier Cabrera Arteaga
KTH Royal Institute of Technology
Stockholm, Sweden
javierca@kth.se

Martin Monperrus
KTH Royal Institute of Technology
Stockholm, Sweden
martin.monperrus@csc.kth.se

Benoit Baudry
KTH Royal Institute of Technology
Stockholm, Sweden
baudry@kth.se

Abstract

The comparison and alignment of runtime traces are essential, e.g., for semantic analysis or debugging. However, naive sequence alignment algorithms cannot address the needs of the modern web: (i) the bytecode generation process of V8 is not deterministic; (ii) bytecode traces are large.

We present STRAC, a scalable and extensible tool tailored to compare bytecode traces generated by the V8 JavaScript engine. Given two V8 bytecode traces and a distance function between trace events, STRAC computes and provides the best alignment. The key insight is to split access between memory and disk. STRAC can identify semantically equivalent web pages and is capable of processing huge V8 bytecode traces whose order of magnitude matches today's web like https://2019.splashcon.org, which generates approx. 150k of V8 bytecode instructions.

CCS Concepts • **Information systems → World Wide Web**; • **Theory of computation → Program semantics**; • **Software and its engineering →** *Interpreters*; *Source code generation*; *Designing software.*

Keywords V8, Sequence alignment, JavaScript, Bytecode, Similarity measurement

ACM Reference Format:
Javier Cabrera Arteaga, Martin Monperrus, and Benoit Baudry. 2019. Scalable Comparison of JavaScript V8 Bytecode Traces. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL '19), October 22, 2019, Athens, Greece.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3358504.3361228

VMIL '19, October 22, 2019, Athens, Greece

ACM ISBN 978-1-4503-6987-9/19/10…$15.00
https://doi.org/10.1145/3358504.3361228

1 Introduction

Runtime traces record the execution of programs. This information captures the dynamics of programs and can be used to determine semantic similarity [29], to detect abnormal program behavior [8], to check refactoring correctness [22] or to infer execution models [1]. In many cases, this is achieved by comparing execution traces, e.g. comparing the traces of the original program and the refactored one. The comparison of program traces can be based on information retrieval [17], tree differencing [9, 27] and sequence alignment [2, 11]. In this paper, we focus on the latter, in order to compare sequences of V8 bytecode instructions resulting from the execution of JavaScript code.

V8 is an open source, high-performance JavaScript engine. For debugging purposes, it provides powerful facilities to export page execution information [21], including intermediate internal bytecode called the V8 bytecode [4].

Due to the dynamic nature of the Web, we observe that the bytecode generation process of V8 is not deterministic. For example, visiting the same page several times results in different V8 bytecode traces every time. This nondeterminism is a key challenge for sequence alignment approaches, even if they perform well on deterministic program traces [10]. Besides, V8 bytecode traces are large. Naive sequence alignment algorithms are time and space quadratic on trace sizes and do not scale to V8 bytecode traces. To illustrate this scaling problem, let us consider a simple query to https://2019.splashcon.org: it generates between 139555 and 162558 V8 bytecode instructions, and aligning two traces of such size, requires approximately 150GB of memory[1]. This memory requirement is not realistic for trace analysis tasks on developer's personal computers or servers. The key challenge that we address in this work is to provide a trace comparison tool that scales to V8 bytecode traces.

In this paper, we present STRAC (Scalable Trace Comparison), a scalable and extensible tool tailored to compare bytecode traces from the V8 JavaScript engine. STRAC implements an optimized version of the DTW algorithm [18]. Given two V8 bytecode traces and a distance function between trace events, STRAC computes and provides the best alignment. The key insight is to split access between memory and disk.

Our experiments compare STRAC with 6 other publicly-available implementations of DTW. The comparison involves

[1] In this paper, memory means RAM.

100 pairs of V8 bytecode traces collected over 6 websites. Our experimental results show that 1) STRAC can identify semantically equivalent web pages and 2) STRAC is capable of processing big V8 bytecode traces whose order of magnitude matches today's web.

To sum up, our contributions are:

- An analysis of the challenges for analyzing browser traces, due to the JavaScript engine internals and the randomness of the environment. We explain and show examples of how the same browser query can generate two different V8 bytecode traces.
- A tool called STRAC that implements the popular alignment algorithm DTW in a scalable way, publicly available at https://github.com/KTH/STRAC .
- A set of experiments comparing 100 V8 bytecode traces collected over 6 real world websites:google.com, kth.se, github.com, wikipedia.org, 2019.splashcon.org and youtube.com. Our experiments show that STRAC copes with the non-deterministic traces and is significantly faster than state-of-the-art tools.

The paper is structured as follows. First we introduce a background of V8 bytecode generation non-determinism and the formalisms used in our work (Section 2). Then follows with technical insights to implement STRAC (Section 3), research question formulation, experimental results with a discussion about them (Section 4). We then present related work (Section 5) and conclude (Section 6).

2 Background

In this section we discuss the key insights behind the non-determinism of the V8 bytecode generation process, as well as the foundations of the DTW alignment algorithm.

2.1 Browser Traces

Our dynamic analysis technique is evaluated with V8 bytecode [19]. In this subsection, we describe how the V8 engine generates bytecode trace. We collect such traces to evaluate our trace comparison tool. In this work, we use the term "V8 bytecode trace" to refer to the result of executing V8 with the *–print-bytecode* flag [21].

2.1.1 V8 Bytecode Generation

The V8 engine compiles JavaScript source code to an intermediate representation called "V8 bytecode". This is done to increase execution performance. The V8 engine parses and compiles every JavaScript code declaration present in HTML pages into a bytecode representation, composed by function declarations, like the one shown in Figure 1. These function declarations came from V8 builtin JavaScript code and external JavaScripts.

V8's bytecode interpreter is a register machine [16]. Figure 1 shows a JavaScript code and its bytecode translation.

```
1  function plusOne(a){ return a.value + 1; }
2  plusOne( {value : 2018} );
```

```
1  [generated bytecode for function: plusOne]
2  Parameter count 2
3  Register count 0
4  Frame size 0
5  30 E> 0x1373c709b6 @ 0 : a5 00 00 00 StackCheck
6  56 S> 0x1373c709b7 @ 1 : 28 02 00 01
       ↪ LdaNamedProperty a0, [0], [1]
7  62 E> 0x1373c709bb @ 5 : 40 01 00 00 AddSmi [1],
       ↪ [0]
8  66 S> 0x1373c709be @ 8 : a9 00 00 00 Return
```

Figure 1. Example of a JavaScript function and its corresponding V8 bytecode instructions.

Each bytecode operator specifies its inputs and outputs as register operands. V8 has 180 different bytecode operators.

The bytecode translation is lazy, i.e. V8 tries to avoid generating code it "thinks" might not be executed. Consequently, a function that is not called will not be compiled [28]. For example, removing line 2 in the top listing of Figure 1 would prevent the compilation of bytecode for the function declared in line 1. This behavior has an impact on the collected traces.

We have observed that V8 bytecode is resilient to script minification and static code-obfuscation techniques. Therefore, we believe that aligning such low-level representations could prove to be a useful aid in many program analysis tasks, such as code similarity study and malware analysis.

2.1.2 Non-Determinism in Browser Traces

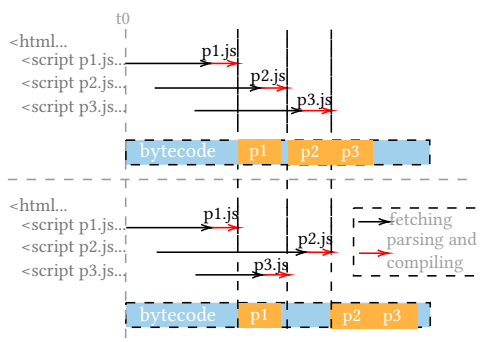


Figure 2. Illustration of two different script fetching and compiling traces for the same browser query.

Interestingly, browsers are fundamentally non deterministic, depending on web server availability, current workload,

and DNS caches through the network. Let us look at the example illustrated in Figure 2. It shows what happens when fetching a web page, which contains 3 scripts. The top and bottom parts illustrate, for the same page, two different executions. Dashed border rectangles represent complete bytecode generation traces. The blue spaces in the bar are V8 common builtin bytecode, which is systematically generated in all browser requests. Orange rectangles illustrate declared page scripts compilations. The complete bytecode trace is the union of both generated bytecodes, builtin V8 and page declared scripts. In the first case at the top of Figure 2, the scripts are fetched and compiled in the same order they are declared. In the second case, at the bottom, **p3.js** is carried and compiled first, before **p2.js** due to a possible network delay. However, V8's compiler will put all scripts compilations in the same order they are declared in the HTML page. The final result is two semantically equivalent bytecode compilations, where script blocks may not be strictly placed in the same position.

The slight differences that occur in the final bytecode for same browser queries motivate us to provide an efficient tool for traces alignment: traces where events occur in different orders but that have the same semantics must be considered as equivalent. The order of events should not confuse the trace comparison tool.

2.1.3 DTW Algorithm

The DTW algorithm has been introduced by Needleman and Wunsch for protein global alignment [18]. Global alignment means trace heads and tails are constrained to match each other in position. DTW is a popular technique for comparing traces in different domains, incl. software traces [14]. DTW finds the best global alignment between two traces, based on a generic similarity function between trace events and gaps.

Definition (Trace) A trace X is defined as a sequence of events. $X = x_1, x_2, ...x_N$ represents a trace of size N where each x_i is the event happening at the ith position.

Definition (Cost Matrix) D is a cost matrix for two traces X and Y of size n and m. D_{ij} stores the optimal cost alignment value for X and Y considered from the start up to the ith and jth positions respectively, that is the minimal cost of aligning x_i and y_j events at the same position in the final alignment.

The cost matrix is defined according to a distance function d and a gap cost γ as follows:

$$D_{0i} = \gamma * i$$

$$D_{j0} = \gamma * j$$

$$D_{ij} = min \begin{cases} D_{i-1j} + \gamma, \\ D_{ij-1} + \gamma \\ D_{i-1j-1} + d(x_i, y_j) \end{cases}$$

In every cell, the value D_{ij} is the minimum cost between putting a gap in one trace and the result of evaluating the distance function between events x_i and y_j.

Definition (Alignment Cost) Given two traces X and Y with sizes N and M respectively, the alignment cost is the value stored in D_{NM}.

Definition (Alignment Difficulty) Given two traces X and Y with sizes N and M respectively, the alignment difficulty is simply the multiplication of both sizes $N \times M$.

Definition (Warp Path) The warp path is the path to go from D_{NM} to the first element D_{00} minimizing the cumulative cost. In general more than one path may exist. Size of warp path is $O(N + M)$.

Definition (Aligned Trace) An aligned trace is a trace where the warp path is applied, i.e. some gaps have been put between some events in one of both traces.

In Figure 3 we illustrate the alignment between traces **abcababc** and **aabaca** with $\gamma = 1$, $d(x_i, y_j) = 2$ if $x_i \neq y_i$ and $d(x_i, y_j) = 0$ if $x_i = y_i$. The warp path is represented as the blue and orange lines going across the matrix from the top left corner to the bottom right corner. In this example, alignment cost is 4, as we can see in bottom right corner cell in Figure 3.

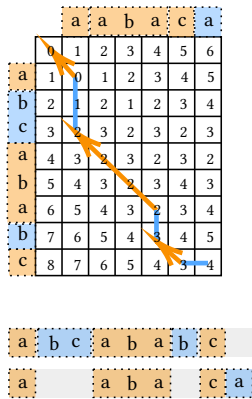


Figure 3. Cost matrix, warp path and applied alignment for **abcababc** and **aabaca** example traces.

3 STRAC: Trace Comparison Tool for V8

STRAC is an approach to compare large traces, tailored to bytecode traces of the V8 JavaScript engine. STRAC takes as input a trace of JavaScript V8 bytecode traces collected in the browser. It produces as output, a trace alignment, and a distance measure between the two traces. STRAC implements the DTW algorithm presented in Subsection 2.1.3. It is an open-source project publicly-available on https://github.com/KTH/STRAC . In this section, we explain the

key components and insights of STRAC to achieve scalable trace comparison.

3.1 Challenges Addressed by STRAC

Non-Determinism As shown in Subsection 2.1.2, V8 can provide two different bytecode traces for the same web page. In this case, both traces are semantically equivalent, but the global position of code modules can vary. These variations occur as a consequence of resource management, interpreter optimizations and JavaScript code fetching from the network. It is challenging because it can provide 1) false positives: two traces may be considered different even when they come from the same pages; 2) false negatives: two traces may be considered the same even when they come from two different pages.

Size Browser traces are huge and naive trace comparison fails on such traces because of memory requirements. For instance, aligning two traces of size 63137 and 58265 events requires a DTW cost matrix, represented as a bidimensional integer matrix, of 14.72 GB of memory. The challenge is to make trace comparison at the scale of browser traces, with tractable memory requirements.

3.2 DTW Distance Functions

The DTW algorithm has two main parameters: a distance function and a gap cost as explained in Subsection 2.1.3. The distance function between events affects the global alignment result, as we show in Subsection 4.5. It defines the matching of two different trace instructions if these instructions have a certain level of similarity. For example, when comparing *'AddSmi [0], [1]'* and *'AddSmi [1], [0]'* instructions, they can be considered as similar because the *AddSmi* operator is in both.

In STRAC, we define two distance functions for bytecode instructions.

$$d_{Sen}(x_i, y_j) = \begin{cases} s \text{ if } x_i \text{ and } y_j \text{ events are exactly} \\ \text{the same bytecode instruction} \\ c \text{ otherwise} \end{cases}$$

$$d_{Inst}(x_i, y_j) = \begin{cases} s \text{ if } x_i \text{ and } y_j \text{ bytecode instructions} \\ \text{share the same bytecode operator} \\ c \text{ otherwise} \end{cases}$$

Both require the identity relationship of the bytecode instruction. For V8 bytecode, based on our results (Subsection 4.5), it seems incoherent to accept an alignment match with two different elements instead of introducing the gap.

We now discuss the value of γ, s and c. The cost of introducing a gap, intuitively, must be less than the cost of matching two different events, i.e. $\gamma < s$. c is the value of matching two equal events, 0. The default values are based on our experience, $s = 5$, $\gamma = 1$ and $c = 0$. The three are configurable.

3.3 Buffering the Cost Matrix

The key limitation of DTW is the need for a large cost matrix to retrieve the warp path. Recall our example requiring 14.72 GB in Subsection 3.1. This means that a naive implementation can only compare small traces due to memory explosion.

In STRAC, we solve this problem by storing the cost matrix both in memory and disk. Only the appropriate values are kept in memory. Our key insight is that the current value D_{ij} in the cost matrix is calculated with the previous row and column, consequently, only $O(N)$ memory space is needed to compute D_{NM}. Thus, STRAC only maintains the current and previous row in memory for each DTW iteration. After processing a row, it is saved to disk. STRAC eventually saves the complete cost matrix to disk.

For traces with lengths 63137 and 58265, instead of 14.72 GB, STRAC requires no more than 86MB of memory for the trace alignment, which represents an improvement of 99.5% in memory consumption.

3.4 Retrieving the Warp Path

In addition to the alignment cost, it is necessary to obtain the warp path in order to create and analyze the aligned traces. Recall that the aligned traces are obtained by applying the warp path on both initial traces, as we mentioned in Subsection 2.1.3.

To retrieve the warp path from the final cost matrix, one goes backward and starts from the trace tail positions (D_{NM}). Cost matrix in D_{ij} depends on three neighbors D_{i-1j}, D_{ij-1} and D_{i-1j-1}. The backtracking process finishes when the trace start is reached, i.e. when the left top corner D_{00} is reached in the matrix. In the warp path construction process, trace indices are always decreasing by one, i.e. trace events are visited only once. Therefore, in STRAC, backtracking over the final cost matrix requires only $O(N + M)$ read operations on disk, which is scalable.

3.5 DTW Approximations

Due to the quadratic time and space complexity of DTW, previous work has proposed approximations to speed up the alignment process. STRAC also implements two state-of-the-art DTW approximations. We now mention these two approximations.

Fixed Regions Using fixed regions is a technique only to evaluate a specified region in the cost matrix [7, 12, 13, 24]. Consequently, the globally optimal warp path will not be found if it is not entirely in the window. This improvement speeds up DTW by a constant factor, but the execution time is still O(NM). STRAC provides support for fixed regions.

FastDTW [2] [25] is an approximation of DTW that has a linear time and space complexity. It combines data abstraction and constraint search in the solution space. STRAC implements FastDTW. Note that, for DTW and its approximations, the default mode is the buffering mode presented in Subsection 3.3.

3.6 Recapitulation

To sum up, STRAC is an optimized implementation of DTW and two approximations with distance functions dedicated to V8 bytecode traces and with neat handling of the cost matrix over memory and disk in order to scale.

4 Experimental Evaluation

We assess the scalability of STRAC for V8 bytecode trace comparison with the following research questions:

- RQ1 (Scalability): To what extent does STRAC scale to traces of real-world web pages?
- RQ2 (Consistency): To what extent does STRAC identify similarity in semantically-equivalent traces?
- RQ3 (Distance Functions): What is the effectiveness of STRAC support of different distance functions?

4.1 Study Subjects

Our experiment is based on tracing the home page of the following sites; google.com, github.com, wikipedia.org, youtube.com, four of the most visited websites, according to Alexa. We also add two sites based on personal interest: 2019.splashcon.org and kth.se, the homepage of our University. All those pages use JavaScript code. The traces were generated just opening the page without any other further action. Since the traces are non-deterministic, we collect 100 traces for the same page. This means we collect 600 traces in total.

Table 1. Descriptive statistics of our benchmark. The 6 sites are sorted by popularity according to the Alexa index. Example bytecodes are available in https://github.com/KTH/STRAC/tree/master/STRACAlign/src/test/resources/bytecodes.

Site	No. scripts	Bytecode size
google.com	5	85768
youtube.com	15	166626
wikipedia.org	4	48260
github.com	3	59384
kth.se	9	64178
2019.splashcon.org	17	147196

Table 1 gives an overview of the collected traces. The first column shows the real world website names. The second and third columns indicate the number of declared scripts and the bytecode size mean value (orange dots in Figure 4) respectively. For instance, Wikipedia loads 4 scripts and produces bytecode traces of 48260 bytecode instructions. This value is the lowest of our benchmark. On the contrary, for Youtube, the page declares 15 JavaScript scripts, and V8 generates traces of 166626 bytecode instructions, and this is due to the richer features of Youtube compared to Wikipedia. In our benchmark, the bytecode traces are in the range of 48k-166k instructions.

Recall that the bytecode traces are non-deterministic even for the same page (see Subsection 2.1.2). We measure how many instructions are contained in each V8 bytecode trace. Figure 4 illustrates the distribution of trace sizes as violin plots. This figure shows that there is a variance of bytecode traces for all pages (Wikipedia also has some variance but this is not shown in the figure because of the scale). This variance is a consequence of several stacked factors: resource management, interpreter optimization and JavaScript code fetching from the network. To our knowledge, this non-determinism in web traces is overlooked by research.

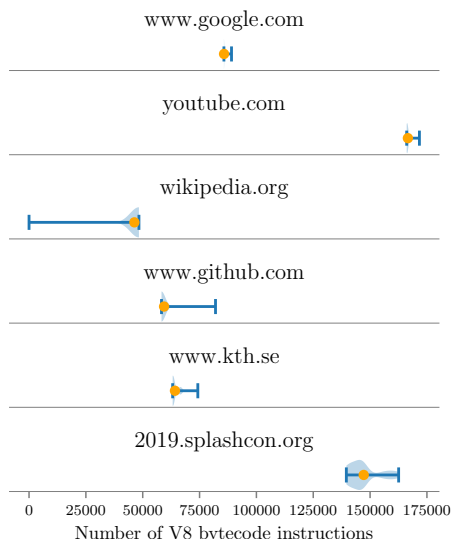


Figure 4. Variance of V8 bytecode trace size for 100 repetitions of the same query.

4.2 Experimental Methodology

Every trace is collected using a non-cached browser session, without plugins. This choice is motivated by two main reasons: 1) we have observed that cached scripts do not affect bytecode generation as direct network fetching does;

[2] The implementation mentioned in the original paper (https://cs.fit.edu/~pkc/FastDTW/) was not available at the moment of this work.

2) browser plugins are compiled to the same bytecode trace and in the scope of this work we are interested only in V8 bytecode traces directly generated from web page scripts.

To answer **RQ1**, we align 12 trace pairs randomly taken from the initial set of all possible trace pairs (600×600). We compare STRAC with different implementations of DTW 1) From public github repositories: rmaestre [3], dtaidistance [4] and pierre-rouanet [5]; 2) From R's dtw package [6] ; 3) The DTW implementation used in [15], slaypni [6]. For each comparison, we compute the average wall-clock execution time.

RQ2 is answered as follows. We select a random sample of 100 pairs from all possible trace pairs (600×600). We select 35 pairs of traces extracted from the same pages and 65 pairs of traces extracted from different pages. Alignment cost is measured for each pair using gap cost $\gamma = 1$ and event distance function d_{Sen} (defined in Subsection 3.2), with parameters: $s = 5$ and $c = 0$. We group and plot each pair alignment cost per site.

We answer **RQ3** using the same traces as *RQ2*. We compute DTW on each one of the 100 sampled pairs. We use the same gap cost $\gamma = 1$, but we compare the two distance functions d_{Sen} and d_{Inst} (defined in Subsection 3.2), with parameters: $s = 5$ and $c = 0$. We measure the alignment cost for each pair and compare the results with the ones obtained in *RQ2*.

The STRAC experimentation has been made on a PC with Intel Core i7 CPU and 16Gb DDR3 of RAM. We extract all traces from Chrome version 74.0.3729.169 (Official Build) (64-bit).

4.3 Answer to RQ1: Scalability

Figure 5 shows the execution time of 6 different alignment tools on 12 trace pairs. The X axis gives the size of the alignment problem, which is the multiplication of the size of both traces in number of bytecode instructions. The Y axis represents the execution time in seconds with a logarithmic scale.

First, we observe that four tools get out of memory for all the considered trace pairs: R-dtw, cpy-wannesm, rmaestre, cpy-slaypul (see the red dot in Figure 5). The main reason for this failure is that those tools need to store the cost matrix in memory. The least difficult trace comparison in the plot is a pair of traces of 48k instructions each. Finding the best alignment for this pair consists in analyzing an eight-bytes integer matrix of approx. 20GB (exactly 18632 millions of bytes). This memory requirement is almost the full memory of modern personal computers and it causes memory explosion at runtime. Applying the same analysis to the most difficult alignment in the plot shows requires 200GB of memory.

[3] https://github.com/rmaestre/FastDTW
[4] https://github.com/wannesm/dtaidistance
[5] https://github.com/pierre-rouanet/dtw
[6] https://github.com/slaypni/fastdtw

Second, py-wannesm and py-pierre-rouanet calculate the best alignment cost for the first 10 pairs, without any memory issue, even for problems in the order of magnitude close to 1.5×10^{10} in alignment difficulty. After this value, these tools also start to get memory issues for the same reason as the other tools. Yet, these succesfully align the 10 pairs (orange and green curves in Figure 5) thanks to an efficient use of Numpy [3] arrays to store cost matrix. Numpy arrays in Python are tailored to efficiently deal with arrays up to 20GB of memory in x64 architectures. We also observe that py-wannesm is always slower than py-pierre-rouanet. The main reason for this time difference is that py-wannesm does an extra pass through the cost matrix and py-pierre-rouanet does not do it.

Third, STRAC succesfully find the best alignment cost for all pairs in the benchmark, even for trace pairs that require memory beyond Numpy capabilities (the last two blue dots in Figure 5). The key insight behind is that STRAC implements the cost matrix data structure as a hybrid between memory and disk, i.e. moving such memory needs to disk.

Both Python implementations (py-wannesm and py-pierre-rouanet) systematically take at least one order of magnitude longer to run, compared to STRAC. The main reason behind this is that Python usually compiles code at runtime, while Java compiles it in advance, making a faster program. Besides, most JVMs perform Just-In-Time compilation to all or part of programs to native code, which significantly improves performance, but mainstream Python does not do this.

Recall that best alignment calculation using naive DTW implementation is non-scalable by its space-time quadratic nature, any implementation of DTW (even the one included in STRAC) eventually will run out of space (in memory or disk) and execution time will be near to impossible. However, STRAC can deal with all trace pairs of our benchmark thanks to its hybrid strategy that leverages both the disk and the memory. To align an average trace of 100k instructions, STRAC takes approx. 14 minutes in a PC like the one mentioned in Subsection 4.2.

4.4 Answer to RQ2: Consistency

In Figure 6, we plot the alignment cost for 100 trace pairs, the blue dots represent pairs extracted from the same page, the orange dots illustrate trace pairs taken from two different pages. Each column corresponds to a given web page. Green dots represent pairs with the maximum alignment cost for each site: an alignment of the web page treated in the column with a trace from the site cited above the dot. For example, the green dot in the first column is an alignment of a trace pair (2019.splashcon, youtube).

In Figure 6, we observe that, for each site, traces from the same page have a lower alignment cost. This is consistent with the fact that in these cases, the majority of both traces

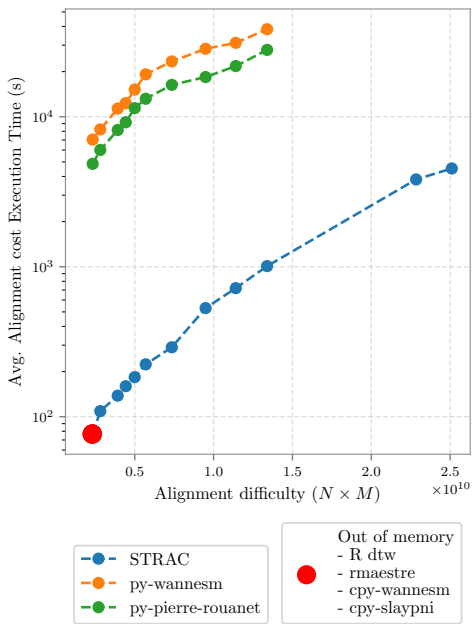


Figure 5. Execution time for **12** trace pair comparisons by 7 tools incl. STRAC. Y axis is in logarithmic scale. Four tools fail even on the smallest traces.

in the pair are the same. On the contrary, the alignment cost between traces from different pages is higher.

Some cases show blue dots with sparsed high values. This occurs when external scripts, declared in some pages, present a high variance in fetching process time. Also, it sometimes happens that for one script declared in a page, the remote servers sends different JavaScript code at each every request. Therefore, the generated bytecode varies more from one load to another, and the alignment cost is increased, showing a small margin between orange dots and the blue ones. However, we observe two scenarios when these phenomena are mitigated. First, when the bytecode generated from the external declaration is larger than the builtin bytecode (2019.splashcon, UNIV, and Youtube cases present a clear separation between clusters). Second, when the fetching process time is stable, as Wikipedia and Github cases show.

In the case of Google, we observe the worst possible scenario. This site has 5 external declared scripts (see Table 1), 3 of them have variable fetching time and their content varies at each load. These 3 scripts integrate Google Analytics features to the site. On the contrary, in the case of Wikipedia, external declared JavaScripts always provide the same code in almost constant time. As a result, the generated bytecode is more deterministic and alignment cost decreases for traces from the same site. In the case of Wikipedia, alignment costs for pairs of traces collected from the same page vary between 1926 and 2652. These values are the lowest alignment costs in the benchmark, and they differ from others in more than 2× in order of magnitude

Overall, the traces from the same (resp. different) page are located in separated clusters. In all cases, we also observe groups of orange dots that can be easily separated from other orange clusters. This separation is a consequence of semantic differences between sites and the increase of JavaScript declarations. For instance, in the first column of Figure 6, trace pairs from 2019.splashcon and Youtube home pages have higher alignment costs. This is a consequence of that Youtube is a richer feature site as 2019.splashcon is, but they semantically differ. We also observe this behavior in the case of Kth and Youtube trace pairs.

V8 compiles builtin JavaScript code to the same bytecode trace, as we discussed in Subsection 2.1.1. This bytecode generation is included in all collected traces. To validate this, we computed the V8 bytecode trace of an empty page: it contains 40k bytecode instructions on average. This also represents a constant noise in the alignment computation.

As Figure 6 illustrates, given the alignment cost of two semantically equivalent traces (blue dots) as a reference, STRAC is capable of identifying similarity with other page traces. However, we want to remark that STRAC accuracy gets improved when JavaScript declarations increase in the compared sites.

4.5 Answer to RQ3: Distance Functions

In Figure 7, we plot the alignment cost using distance d_{Ins}. Recall that d_{Ins} is less restrictive than d_{Sen}, the distance used to answer **RQ2**. By comparing Figure 7 and Figure 6, we observe interesting phenomena. First, changing the distance function breaks the clustering breakdown for Github, Google and Kth (some blue points get mixed with orange points). Second, the maximum alignment cost is lower than in Figure 6 for all sites. These phenomena are consequences of using a less restrictive distance function, i.e. with d_{Ins}, only the operator is analyzed in the bytecode instructions comparison. Overall, the choice of distance function matters. STRAC can be extended with new distance functions and provides d_{Sen} by default for properly aligning V8 bytecode traces.

We notice that the impact of the distance function is bigger for sites with less JavaScript. For Google, Github and Wikipedia, using d_{Ins} is bad because it breaks the clustering. For the remaining three websites, which involve more JavaScript features, while the alignment changes, the core property of the alignment of identifying semantically equivalent traces still holds.

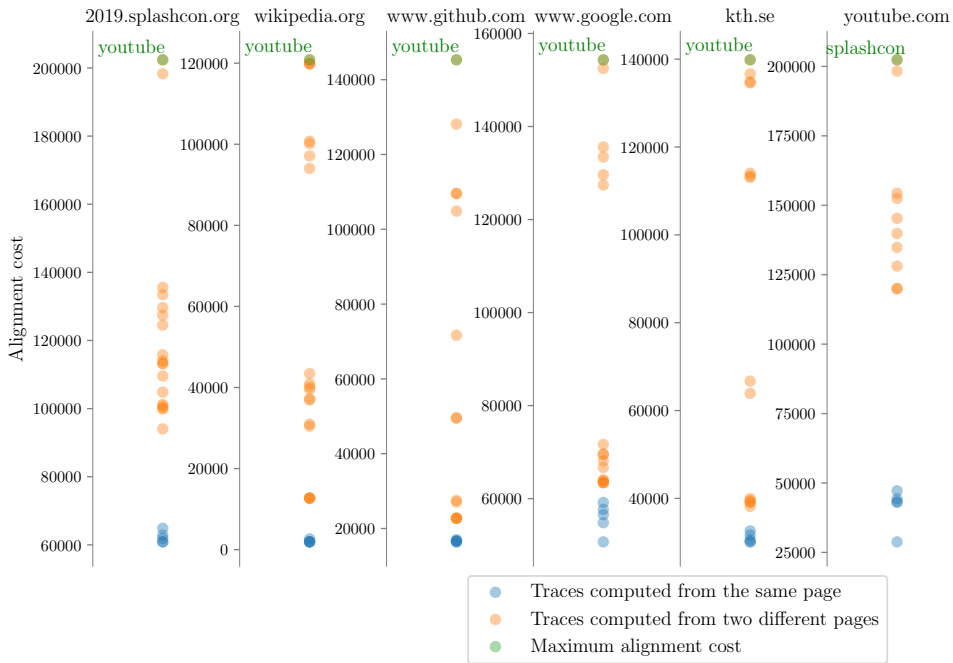


Figure 6. Alignment costs for 100 trace pair comparisons using d_{Sen} as distance function.

5 Related Work

DTW is memory greedy on trace size, a similar problem arises when dealing with streaming traces. Oregi et al. [20] and Martins et al. [15] present a generalization of DTW for large streaming data. They propose the use of incremental computation of the cost matrix complemented with a weighted event distance function adding event positions. However, their results may differ from the original DTW warp path. On the contrary, STRAC also computes the exact alignment cost without approximations.

Kargen et al. [10] propose a combination of data abstraction and FastDTW to align two program traces at the binary level. They record and analyze read and write operations to memory and x86 registers. Also, they argue and they show that their method scales to large traces. STRAC is also capable of analyzing such traces, but targets different kinds of traces: V8 bytecode traces, which are not handled by Kargen et al.

Ratanaworabhan et al. [23] instrument Internet Explorer to measure JavaScript runtime and static behavior in function calls and event handlers on real-world websites. By doing so, they show that common benchmarks, like SpiderMonkey and V8-Suite, are not representative of real application behavior. We could use STRAC to perform a similar analysis on modern browsers.

With JALANGI, Sen et al. [26] provide a framework to dynamically analyze JavaScript. The framework works through source code instrumentation. JALANGI associates shadow values to variables and objects in the instrumented code, Sen et al. argue that most of of state-of-the-art dynamic analysis techniques can be implemented, like concolic evaluation and taint analysis. However, JALANGI has several limitations dealing with builtin code and instrumentation can decrease instrumented code execution performance. With STRAC, we propose to use V8 bytecode traces to compare JavaScript semantic similarity without JavaScript instrumentation.

Fang et al. [5] propose a JavaScript malicious code detection model based on neural networks. To mitigate the obfuscation techniques used in malicious code, they analyze the dynamic information recorded in V8 bytecode traces. Both STRAC and Fang et al. consider V8 bytecode traces, yet the usages are different: they do anomaly detection while we do trace comparison.

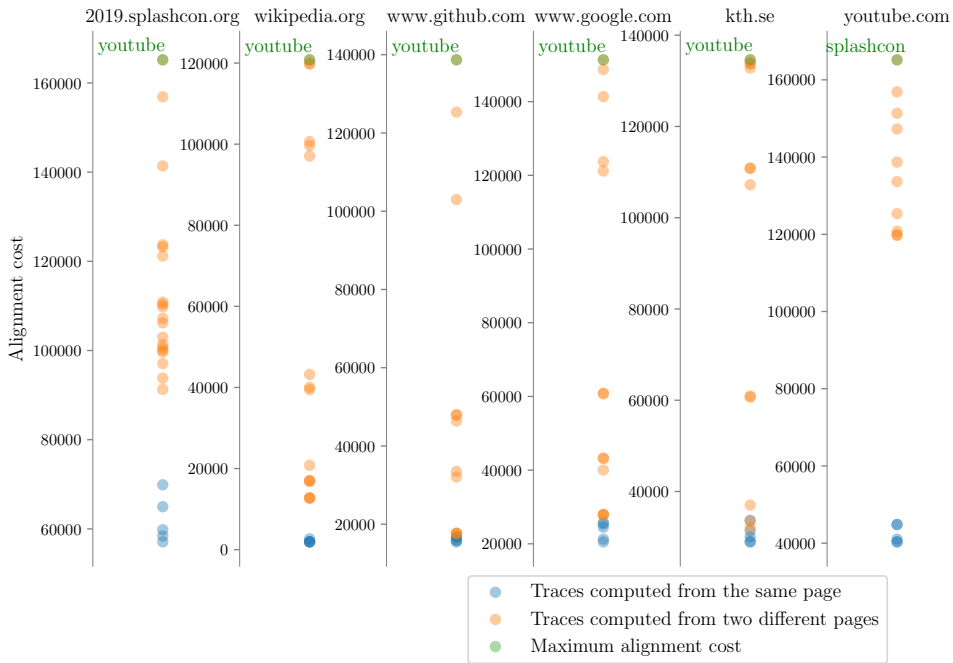


Figure 7. Alignment cost for 100 trace pair comparisons using d_{Ins} as distance function.

6 Conclusion

In this paper, we presented a tool, called STRAC, for aligning execution traces. STRAC is tailored to traces of the JavaScript V8 engine. STRAC implements an optimized version of the DTW algorithm and two of its approximations. Our experiments show that STRAC scales to real-world JavaScript traces consisting of V8 bytecodes. STRAC provides two distance functions for trace event comparison and can be configured with any arbitrary distance function. Our evaluation indicates that STRAC performs better than state of the art DTW implementations, for 6 representative web sites.

We have shown that V8 bytecode contains redundancy and that an empty page includes more than 40k trace instructions. By removing this redundant and useless trace instructions, the alignment would get better. In our future work, we will study how to remove redundancy in V8 bytecode traces, for providing a better behavioral similarity measure for modern web pages full of JavaScript code.

Acknowledgments

This material is based upon work supported by the Swedish Foundation for Strategic Research under the Trustfull project and by the Wallenberg Autonomous Systems and Software Program (WASP).

References

[1] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D. Ernst. 2011. Leveraging Existing Instrumentation to Automatically Infer Invariant-Constrained Models. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering - SIGSOFT/FSE '11* (2011). ACM Press, 267. https://doi.org/10.1145/2025113.2025151

[2] Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. 2019. Semantic Program Alignment for Equivalence Checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. ACM, New York, NY, USA, 1027–1040. https://doi.org/10.1145/3314221.3314596

[3] Numpy community. 2018. Numeric python. https://www.numpy.org/index.html

[4] V8 JavaScript engine. 2016. *Ignition design documentation.* https://v8.dev/docs/ignition

[5] Y. Fang, C. Huang, L. Liu, and M. Xue. 2018. Research on Malicious JavaScript Detection Technology Based on LSTM. *IEEE Access* 6 (2018), 59118–59125. https://doi.org/10.1109/ACCESS.2018.2874098

[6] Toni Giorgino. 2009. Computing and Visualizing Dynamic Time Warping Alignments in R: The dtw Package. *Journal of Statistical Software, Articles* 31, 7 (2009), 1–24. https://doi.org/10.18637/jss.v031.i07

[7] F. Itakura. 1975. Minimum prediction residual principle applied to speech recognition. *IEEE Transactions on Acoustics, Speech, and Signal Processing* 23, 1 (February 1975), 67–72. https://doi.org/10.1109/TASSP.1975.1162641

[8] G. Jiang, H. Chen, C. Ungureanu, and K. Yoshihira. 2007. Multiresolution Abnormal Trace Detection Using Varied-Length *n*-Grams and Automata. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 37, 1 (Jan 2007), 86–97. https://doi.org/10.1109/TSMCC.2006.871569

[9] T. Kamiya. 2018. Code difference visualization by a call tree. In *2018 IEEE 12th International Workshop on Software Clones (IWSC)*. 60–63. https://doi.org/10.1109/IWSC.2018.8327321

[10] Ulf Kargén and Nahid Shahmehri. 2017. Towards Robust Instruction-level Trace Alignment of Binary Code. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 342–352. http://dl.acm.org/citation.cfm?id=3155562.3155608

[11] Hyunjoo Kim, Jonghyun Kim, Youngsoo Kim, Ikkyun Kim, Kuinam J. Kim, and Hyuncheol Kim. 2017. Improvement of malware detection and classification using API call sequence alignment and visualization. *Cluster Computing* (12 Sep 2017). https://doi.org/10.1007/s10586-017-1110-2

[12] Daniel Lemire. 2008. Faster Retrieval with a Two-Pass Dynamic-Time-Warping Lower Bound. *CoRR* abs/0811.3301 (2008). arXiv:0811.3301 http://arxiv.org/abs/0811.3301

[13] Y. Lou, H. Ao, and Y. Dong. 2015. Improvement of Dynamic Time Warping (DTW) Algorithm. In *2015 14th International Symposium on Distributed Computing and Applications for Business Engineering and Science (DCABES)*. 384–387. https://doi.org/10.1109/DCABES.2015.103

[14] Marcelo De A. Maia, Victor Sobreira, Klérisson R. Paixão, Ra A. De Amo, and Ilmério R. Silva. 2008. Using a sequence alignment algorithm to identify specific and common code from execution traces. In *Proceedings of the 4th International Workshop on Program Comprehension through Dynamic Analysis (PCODA*. 6–10.

[15] R. M. Martins and A. Kerren. 2018. Efficient Dynamic Time Warping for Big Data Streams. In *2018 IEEE International Conference on Big Data (Big Data)*. 2924–2929. https://doi.org/10.1109/BigData.2018.8621878

[16] Ross McIlroy. 2016. Ignition: V8 Interpreter. https://docs.google.com/document/d/11T2CRex9hXxoJwbYqVQ32yIPMh0uouUZLdyrtmMoL44/edit

[17] L. Moreno, J. J. Treadway, A. Marcus, and W. Shen. 2014. On the Use of Stack Traces to Improve Text Retrieval-Based Bug Localization. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 151–160. https://doi.org/10.1109/ICSME.2014.37

[18] Saul B. Needleman and Christian D. Wunsch. 1970. A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins. 48, 3 (1970), 443–453. https://doi.org/10.1016/0022-2836(70)90057-4

[19] V8 official web page. 2019. *V8 JavaScript Engine*. https://v8.dev/

[20] Izaskun Oregi, Aritz Pérez, Javier Del Ser, and José A. Lozano. 2017. On-Line Dynamic Time Warping for Streaming Time Series. In *Machine Learning and Knowledge Discovery in Databases*, Michelangelo Ceci, Jaakko Hollmén, Ljupco Todorovski, and Saso Vens, Celinand Dzeroski (Eds.). Springer International Publishing, Cham, 591–605.

[21] The Chromium Projects. 2019. *Run Chromium with Flags - The Chromium Projects*. https://www.chromium.org/developers/how-tos/run-chromium-with-flags#TOC-V8-Flags

[22] David A Ramos and Dawson R. Engler. 2011. Practical, Low-effort Equivalence Verification of Real Code. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)*. Springer-Verlag, Berlin, Heidelberg, 669–685. http://dl.acm.org/citation.cfm?id=2032305.2032360

[23] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin G. Zorn. 2010. JSMeter: Comparing the Behavior of JavaScript Benchmarks with Real Web Applications. In *Proceedings of the 2010 USENIX Conference on Web Application Development (WebApps'10)*. USENIX Association, Berkeley, CA, USA, 3–3. http://dl.acm.org/citation.cfm?id=1863166.1863169

[24] H. Sakoe and S. Chiba. 1978. Dynamic programming algorithm optimization for spoken word recognition. *IEEE Transactions on Acoustics, Speech, and Signal Processing* 26, 1 (February 1978), 43–49. https://doi.org/10.1109/TASSP.1978.1163055

[25] Stan Salvador and Philip Chan. 2007. FastDTW: Toward Accurate Dynamic Time Warping in Linear Time and Space. *Intell. Data Anal.* 11, 5 (Oct. 2007), 561–580. http://dl.acm.org/citation.cfm?id=1367985.1367993

[26] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: A Selective Record-replay and Dynamic Analysis Framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, New York, NY, USA, 488–498. https://doi.org/10.1145/2491411.2491447

[27] Ryo Suzuki, Gustavo Soares, Andrew Head, Elena Glassman, Ruan Reis, Melina Mongiovi, Loris D'Antoni, and Bjoern Hartmann. 2017. TraceDiff: Debugging Unexpected Code Behavior Using Trace Divergences. *CoRR* abs/1708.03786 (2017). arXiv:1708.03786 http://arxiv.org/abs/1708.03786

[28] Toon Verwaest and Marja Hölttä. 2019. *Blazingly Fast Parsing, Part 2: Lazy Parsing · V8*. https://v8.dev/blog/preparser

[29] M. Weber, R. Brendel, and H. Brunst. 2012. Trace File Comparison with a Hierarchical Sequence Alignment Algorithm. In *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*. 247–254. https://doi.org/10.1109/ISPA.2012.40